

Meu Primeiro Livro de Python



Eduardo Corrêa
Versão 2.0.0 - 26/02/2020

Detalhes de Copyright



Este trabalho está licenciado com uma Licença [Creative Commons - Atribuição-NãoComercial 4.0 Internacional](https://creativecommons.org/licenses/by-nc/4.0/). Isto significa que você pode alterar, transformar ou criar outra obra com base nesta, contanto que atribua o crédito ao autor original e não utilize a obra derivada para fins comerciais.

Versões

11/02/2019 – primeira versão completa do livro (v 1.0.0)

22/02/2019 – incorporação da primeira errata (v 1.0.1)

02/03/2020 – segunda versão completa do livro (v 2.0.0), com revisão geral do texto e a inclusão dos seguintes tópicos: funções lambda, listas 2d, orientação a objetos e módulo ‘datetime’.

Número ISBN: 978-65-900095-0-0

Título: Meu primeiro livro de Python

Autor: E. Corrêa

Palavras-chave: python, ciência de dados, numpy, pandas

Tipo de Suporte: Publicação digitalizada

Formato Ebook: PDF

edubd 2020

Este livro é dedicado a Glauce, meu grande amor.

Sumário

Propósito.....	7
Agradecimentos.....	10
Capítulo I. Muito Prazer, Linguagem Python.....	11
Lição 1 – O que é Python?.....	12
Lição 2 – Distribuições do Python.....	14
Lição 3 – Instalação do WinPython.....	15
Lição 4 – Utilizando o WinPython no Modo Interativo.....	18
Lição 5 – Utilizando o WinPython no Modo Script.....	21
Lição 6 – Programação Python: Primeiros Passos.....	23
Lição 7 – Instruções de Desvio Condicional: if, else, elif.....	28
Lição 8 – Instruções de Repetição (1): while.....	33
Lição 9 – Instruções de Repetição (2): for – range().....	36
Lição 10 – Python ou R?.....	39
Capítulo II. Criando e Utilizando Funções.....	41
Lição 11 – Criando Funções.....	42
Lição 12 – Funções Pré-Definidas.....	48
Lição 13 – Funções <i>lambda</i>	49
Lição 14 – Módulo ‘math’.....	50
Lição 15 – Módulo ‘statistics’.....	53
Lição 16 – Módulo ‘random’.....	55
Lição 17 – Função print().....	56
Lição 18 – Função help().....	57
Capítulo III. Estruturas de Dados Nativas.....	59
Lição 19 – Listas.....	60
Lição 20 – Listas Bidimensionais.....	68
Lição 21 – List Comprehension.....	71
Lição 22 – Tuplas.....	72
Lição 23 – Tuple Assignment.....	73
Lição 24 – Conjuntos.....	73
Lição 25 – Dicionários.....	75
Capítulo IV. Strings e Bases de Dados no Formato Texto.....	80
Lição 26 – <i>Strings</i>	81
Lição 27 – Arquivos Texto: File Handle.....	86
Lição 28 – Arquivos Texto: Processando Arquivos Separado por Colunas.....	87
Lição 29 – Arquivos Texto: Importando um Arquivo Inteiro para uma String.....	89
Lição 30 – Arquivos Texto: Processando Arquivos CSV.....	90
Lição 31 – Arquivos Texto: Gravando Arquivos.....	92
Lição 32 – Arquivos Texto: Conhecendo o Padrão Unicode.....	93
Lição 33 – Arquivos Texto: Módulo ‘csv’.....	95
Lição 34 – Arquivos Texto: UTF-8 versus ANSI.....	98
Lição 35 – Arquivos Texto: Processando Arquivos JSON.....	100
Lição 36 – Expressões Regulares.....	104
Capítulo V. Banco de Dados e Linguagem SQL.....	111
Lição 37 – Banco de Dados.....	112
Lição 38 – SQL: SELECT Básico.....	114
Lição 39 – SQL: Junção de Tabelas.....	119

Lição 40 – SQL: Produzindo Resultados Agregados.....	123
Capítulo VI. Biblioteca ‘NumPy’.....	126
Lição 41 – NumPy: Introdução.....	127
Lição 42 – NumPy: Criação de Vetores.....	128
Lição 43 – NumPy: Tipos de Dados.....	132
Lição 44 – NumPy: Criação de Matrizes.....	133
Lição 45 – NumPy: Iteração.....	137
Lição 46 – NumPy: Operações Aritméticas.....	137
Lição 47 – NumPy: Fatiamento de Matrizes.....	140
Lição 48 – NumPy: Um Pouco de Matemática e Estatística.....	145
Lição 49 – NumPy: Álgebra Linear.....	151
Lição 50 – NumPy: Ordenação.....	152
Capítulo VII. Biblioteca ‘pandas’.....	153
Lição 51 – pandas: Introdução.....	154
Lição 52 – pandas: Estrutura Series.....	155
Lição 53 – pandas: Introdução à Estrutura DataFrame.....	159
Lição 54 – pandas: Importando Arquivos para Data Frames.....	162
Lição 55 – pandas: Transformação de DataFrames – Conceitos Básicos.....	170
Lição 56 – pandas: Fatiamento de DataFrames.....	174
Lição 57 – pandas: Funções Estatísticas em DataFrames.....	176
Lição 58 – pandas: Junção e Concatenação de DataFrames.....	178
Lição 59 – pandas: Produzindo Resultados Agregados.....	183
Lição 60 – pandas: Trabalhando com Arquivos Grandes.....	184
Capítulo VIII. Biblioteca ‘Matplotlib’.....	188
Lição 61 – Matplotlib: Introdução.....	189
Lição 62 – Matplotlib: Gráfico de Linha.....	190
Lição 63 – Matplotlib: Configurações Básicas.....	191
Lição 64 – Matplotlib: Gráfico de Barras.....	192
Lição 65 – Matplotlib: Gráfico de Dispersão.....	194
Lição 66 – Matplotlib: Histograma.....	195
Lição 67 – Matplotlib: Boxplot.....	197
Lição 68 – Matplotlib: Configurando Subplots.....	197
Lição 69 – Matplotlib: Salvando um Gráfico.....	198
Capítulo IX. Introdução à Programação Orientada a Objetos.....	200
Lição 70 – Classes e Objetos.....	201
Lição 71 – Definindo Classes.....	201
Lição 72 – Instanciando Objetos.....	202
Lição 73 – Consultando e Modificando Atributos.....	204
Lição 74 – Atributos de Classe.....	205
Lição 75 – Herança.....	206
Capítulo X. Manipulação de Dados do Tipo Data/Hora.....	208
Lição 76 – Módulo ‘datetime’.....	209
Lição 77 – Operações sobre Datas.....	209
Lição 78 – Operações sobre DateTimes.....	211
Lição 80 – strftime().....	213
Lição 81 – Classe timedelta.....	214
Lição 82 – Utilizando o from ... import.....	215
Anexo A - Temas Sugeridos para Estudo.....	216

Anexo B - CPython.....	218
Bibliografia.....	223
Sobre o Autor.....	225

Propósito

Ciência de dados (*data science*) é a disciplina que combina ideias da Estatística e da Ciência da Computação para resolver o problema da descoberta de conhecimento em bases de dados. Nesta parceria, a Estatística tem o papel de fornecer as ferramentas para descrever, analisar, resumir, interpretar e realizar inferências sobre os dados. Por sua vez, a Ciência da Computação preocupa-se em oferecer tecnologias eficientes para o armazenamento, acesso, integração e transformação dos dados. Ou seja, o papel da Ciência da Computação é tornar viável a análise de bases de dados, muitas vezes complexas e volumosas, através de processos estatísticos. Dentre as diferentes tecnologias utilizadas para computação científica, **Python** é, sem dúvida, uma das que alcançou maior destaque. Trata-se de uma linguagem de programação livre, extremamente versátil e poderosa, que tem sido largamente adotada em projetos relacionados à ciência de dados, tanto pela indústria quanto pela comunidade acadêmica.

Este livro apresenta os conceitos e técnicas fundamentais para aqueles que desejam começar a trabalhar com Python para ciência de dados. O livro cobre os **aspectos computacionais** da ciência de dados, o que significa que tem como foco principal ensinar o leitor a desenvolver programas capazes de processar bases de dados de diferentes tamanhos, formatos e graus de complexidade. O trabalho destina-se a todo tipo de profissional envolvido com ciência de dados: biólogos, matemáticos, engenheiros, cientistas sociais, químicos, administradores, físicos, estatísticos, economistas, etc., enfim, qualquer pessoa que deseje aprender a desenvolver os seus próprios *scripts* Python para explorar bases de dados relacionadas a problemas de sua área de atuação. Reiterando: o livro não é destinado apenas a pessoas com formação em computação, mas sim a todo e qualquer ser humano interessado em Python para ciência de dados. É ainda importante deixar claro que o livro não tem como foco o ensino da estatística, aprendizado de máquina (*machine learning*) ou mineração de dados (*data mining*). Na realidade, o que pretendemos é ensinar o leitor a **programar na linguagem Python**, capacitando-o a, futuramente, desenvolver qualquer tipo de *script* nesta linguagem, incluindo programas que sejam capazes de analisar grandes bases de dados através de métodos estatísticos ou utilizando algoritmos de aprendizado de máquina e mineração de dados. Em resumo, o que desejamos é tornar o leitor um *pythonista*¹ de primeira linha!

Não é necessário nenhum pré-requisito para a leitura do livro, embora o conhecimento de alguma linguagem de programação – como R, MATLAB, C ou, até mesmo, a programação de macros do Excel – certamente ajude a acelerar o processo de aprendizado. O livro está dividido em dez capítulos. Os três primeiros cobrem o “arroz com feijão” da linguagem, isto é, o mínimo que você precisa saber para começar a desenvolver qualquer tipo de aplicação em Python. Já os capítulos seguintes tratam de temas que possuem relação mais direta com a ciência de dados.

- Capítulo 1, Muito Prazer, Linguagem Python. Tem por objetivo apresentar o ambiente Python e ensinar o leitor a criar os seus primeiros programas.
- Capítulo 2, Criando e Utilizando Funções. A análise de dados é sempre facilitada com o uso de funções. Neste capítulo você descobrirá como utilizar as funções matemáticas e estatísticas básicas do Python e também aprenderá a criar as suas próprias funções e módulos reutilizáveis.

¹ Este é o termo mundialmente utilizado para designar os programadores Python

- Capítulo 3. Estruturas de Dados Nativas. As estruturas de dados são utilizadas pelas linguagens de programação para organizar conjuntos de dados relacionados em memória, visando tornar sua manipulação mais simples e eficiente. Este capítulo apresenta as quatro estruturas de dados nativas do Python: listas, tuplas, conjuntos e dicionários.
- Capítulo 4. Strings e Bases de Dados no Formato Texto. Antes que os dados possam ser analisados por técnicas estatísticas, eles precisam ser carregados no ambiente Python. Este capítulo apresenta as técnicas básicas para importar bases de dados texto estruturadas de diferentes maneiras: CSV, JSON, arquivo separado por colunas, etc. Além disso, o capítulo apresenta as numerosas ferramentas para processamento de texto oferecidas pelo Python, desde as simples funções de string até as expressões regulares.
- Capítulo 5. Banco de Dados e Linguagem SQL. O objetivo principal deste capítulo é ensinar você a consultar, combinar e explorar tabelas armazenadas em bancos de dados relacionais utilizando a linguagem SQL. Embora a SQL tenha mais de 35 anos de existência, ela continua muito relevante, sendo atualmente considerada uma das tecnologias-chave na área de ciência de dados.
- Capítulo 6. Biblioteca 'NumPy'. A biblioteca 'NumPy' (*Numerical Python*) estende a linguagem Python com a estrutura de dados ndarray, voltada para a computação de alto desempenho sobre vetores e matrizes. A 'NumPy' é a biblioteca fundamental para computação científica em Python, uma vez que quase todas as demais bibliotecas a utilizam como base. Neste capítulo, você aprenderá as principais técnicas para criar e processar ndarrays.
- Capítulo 7. Biblioteca 'pandas'. Capítulo que tem por objetivo principal introduzir a biblioteca 'pandas' (*Python Data Analysis Library*), a mais importante dentre todas as ferramentas Python que são voltadas para ciência de dados. Esta biblioteca foi especialmente projetada para oferecer suporte às atividades de seleção, exploração, integração, limpeza e transformação de bases de dados. De fato, a pandas é tão poderosa e sofisticada que precisei escrever um outro livro inteiro apenas sobre ela, denominado "Pandas Python: Data Wrangling para Ciência de Dados", publicado pela editora Casa do Código (<https://www.casadocodigo.com.br/products/livro-pandas-python>). No presente livro, apresentamos uma introdução às funcionalidades básicas das Series e DataFrames, as duas estruturas de dados oferecidas pela pandas.
- Capítulo 8. Biblioteca 'Matplotlib'. Gráficos são ferramentas importantes não apenas para apresentar resultados, mas também quando você está realizando o estudo preliminar de uma determinada base de dados. Neste capítulo, oferecemos uma breve introdução ao pacote 'Matplotlib', destinado à criação de diferentes tipos gráficos no ambiente Python.
- Capítulo 9. Introdução à Programação Orientada a Objetos. Para os cientistas de dados que desejam estudar o código-fonte de bibliotecas *open source* ou para aqueles que possuam a ambição de criar os seus próprios pacotes é muito importante conhecer o paradigma da programação orientada a objetos (POO). Este capítulo tem o objetivo de servir como uma porta de entrada ao universo da POO.

- Capítulo 10. Datas e Horas. Muitas aplicações de ciência de dados exigem a manipulação de informações do tipo data/hora. Neste capítulo apresentamos os recursos oferecidos pela biblioteca ‘datetime’ facilitar o tratamento desse tipo de informação.

Grande parte do texto apresentado neste livro refere-se a código Python (pequenos programas que demonstram os conceitos apresentados) e os resultados produzidos pelo código (normalmente na forma de uma saída impressa na tela). As seguintes convenções tipográficas foram adotadas:

- `fonte com largura constante`
 - Usada nas listagens dos programas Python e na apresentação do conteúdo de bases de dados estruturadas em arquivos texto.
- **fonte com largura constante em negrito**
 - Usada para representar os nomes das palavras reservadas, funções e alguns operadores da linguagem Python. Esta convenção é adotada tanto nas listagens dos programas como nos textos que explicam o funcionamento dos mesmos.
- “palavra entre aspas duplas”
 - Usada nos textos explicativos para destacar nomes de variáveis, arquivos e outros objetos (DataFrames, arrays, tabelas de banco de dados, etc.).

Por fim, é importante mencionar que este **livro digital** é acompanhado de um arquivo chamado “CursoPython.ZIP”, que contém **todos** os programas e bases de dados apresentados ao longo dos capítulos. Mais especificamente, o arquivo possui 30 bases de dados e 110 programas, que você poderá abrir, analisar, alterar, testar e executar de uma maneira simples e rápida no computador de sua casa ou trabalho. Tanto o livro digital quanto o arquivo zip podem ser obtidos em qualquer um dos seguintes endereços:

- Repositório do livro no GitHub:
 - https://github.com/edubd/meu_primeiro_livro_de_python
- Página do livro no ResearchGate:
 - https://www.researchgate.net/publication/331287633_Meu_Primeiro_Livro_de_Python

Agradecimentos

Para iniciar, gostaria de agradecer a toda minha família: Joaquim, Ana, Glauce, Amanda, Inês e Antonio. Sem vocês eu não teria tanta tranquilidade para programar e pesquisar! Um agradecimento especial também para as duas instituições que mais contribuíram e continuam contribuindo para o meu desenvolvimento profissional: Universidade Federal Fluminense (UFF) e Escola Nacional de Ciências Estatísticas (ENCE).

Gostaria de aproveitar para mencionar alguns professores com os quais trabalhei e que admiro muito, por serem não apenas grandes pesquisadores, mas também craques da didática: Alexandre Plastino (UFF), Alex Freitas (University of Kent) e Simone Martins (UFF). E um agradecimento muito especial para o professor Alexandre Federici (ENCE) que me emprestou a sala e o computador onde escrevi grande parte deste livro.

Capítulo I. Muito Prazer, Linguagem Python

Este capítulo tem por objetivo realizar uma apresentação do ambiente Python, assim como ensinar você a escrever os seus primeiros programas na linguagem. Ele começa explicando o que é Python e discutindo os motivos pelos quais essa tecnologia se tornou fundamental para a ciência de dados. Em seguida, apresenta o roteiro passo a passo para a instalação, configuração e utilização do WinPython, um dos mais simples ambientes para programação científica em Python. A partir daí, o capítulo começa a abordar a programação propriamente dita, através de uma sequência de lições que têm por objetivo introduzir os recursos básicos da linguagem Python: variáveis, operadores aritméticos, entrada e saída, estruturas de desvio, estruturas de repetição e blocos de código. Fechando o capítulo, apresentamos uma breve comparação entre Python e R, as duas tecnologias que atualmente rivalizam pelo posto de linguagem de programação mais importante para ciência de dados.



Lição 1 – O que é Python?

Python é uma linguagem de programação de **propósito geral**, o que significa que ela pode ser empregada nos mais diferentes tipos de projetos, variando desde aplicações Web até sistemas de inteligência artificial. A linguagem foi criada no ano de 1991, tendo como principal filosofia priorizar a construção de programas simples e legíveis (aquilo que os *pythonistas* chamam de programas “bonitos”) sobre a construção de programas que, ainda que velozes, sejam complicados e pouco legíveis (os ditos programas “feios”). No decorrer dos dez anos seguintes, a linguagem alcançou grande popularidade tanto em ambiente acadêmico como corporativo. Isto motivou o surgimento da *Python Software Foundation* no ano de 2001, uma instituição independente e sem fins lucrativos que tornou-se responsável pelo desenvolvimento de novas versões da linguagem (no momento da elaboração desta edição do livro, a versão mais recente era a 3.8.1).

Nos últimos anos, Python consolidou-se como uma das tecnologias mais difundidas na área ciência de dados, apesar de não ter sido originalmente projetada para este fim. Essa conquista não se deveu apenas ao fato de a linguagem facilitar a criação de programas de “rosto bonito”. Na realidade, o sucesso do Python para ciência de dados está relacionado a outras de suas características, conforme apresenta-se a seguir:

- Como Python é uma linguagem **interpretada**, os iniciantes podem aprender alguns comandos e começar a fazer coisas legais (ex.: aplicar funções matemáticas e estatísticas sobre conjuntos de dados) quase que imediatamente, sem esbarrar em problemas relacionados à compilação de código. E para tornar a coisa ainda melhor, o interpretador Python pode ser utilizado de forma **interativa**, onde cada comando digitado é imediatamente traduzido e executado. Isto oferece aos programadores uma forma ágil e simples para examinar em tempo real os resultados intermediários obtidos em qualquer passo de um processo de análise de dados.
- Python é uma **linguagem livre** (*open source*). No Web site da *Python Software Foundation*² é possível baixar gratuitamente o arquivo que instala o interpretador Python e a sua biblioteca padrão (a famosa **standard library**). Juntos, estes componentes formam o “coração” do ambiente Python, oferecendo um rico conjunto de estruturas de dados (como listas e dicionários) e centenas de módulos voltados para a execução dos mais diversos tipos de tarefas, desde o uso de funções matemáticas e estatísticas até o processamento de arquivos texto em diferentes formatos (CSV, JSON, etc.).
- A linguagem Python pode ser facilmente estendida através da incorporação de outros **pacotes**. Atualmente, existem milhares de pacotes disponíveis no repositório central do Python (*Python Package Index* – PyPI). Muitos e muitos deles são voltados para ciência de dados, tais como, ‘NumPy’ (manipulação de vetores e matrizes), ‘SciPy’ (rotinas numéricas para resolver problemas de integração, equações algébricas e cálculo numérico, entre outras coisas), ‘pandas’ (importação e transformação de bases de dados), ‘Matplotlib’ (geração de gráficos) e ‘scikit-learn’ (algoritmos de mineração de dados e aprendizado de máquina).

2 <https://www.python.org/>

- Por fim, Python é uma linguagem **multiparadigma**, que permite tanto a elaboração de programas no estilo **procedural** como no **orientado a objetos** (Python também suporta o paradigma funcional, mas este não é coberto no presente livro).
 - O paradigma procedural é mais simples, porém perfeito para o desenvolvimento de *scripts* de ciência de dados. Isso porque ele não exige do programador o domínio de certos conceitos que são muito específicos da área de engenharia de software³ (como, por exemplo, a definição de classes e métodos e a elaboração de rotinas detalhadas para tratamento de exceções). Esta característica do estilo procedural é muito importante na área de ciência de dados, onde frequentemente os programadores não são pessoas com formação em computação, mas sim biólogos, estatísticos, químicos, etc., que desejam apenas criar um programa bem enxuto e direto para explorar uma determinada base de dados.
 - Por outro lado, o paradigma orientado a objetos é sofisticado e complexo, sendo, no entanto, o mais indicado para a construção de aplicativos e sistemas de informação. Em programas deste tipo, o desenvolvedor precisa se preocupar com uma série de questões que nem sempre fazem parte do mundo da análise de dados, tais como o gerenciamento de sessões, o controle de acesso e a criação de menus, apenas para citar algumas poucas. É exatamente no paradigma orientado a objetos que são oferecidas as ferramentas mais adequadas para lidar com estes problemas. Porém, vale a pena observar que um pouco de conhecimento sobre orientação a objetos é importante para qualquer cientista de dados, pelo fato de este paradigma ser o mais recomendado para a construção de **pacotes**.

Muito Prazer, Linguagem Python

- Python é uma linguagem de propósito geral, interpretada, interativa, gratuita e multiparadigma.
- Trata-se de uma linguagem extremamente versátil e poderosa que possui um grande número de pacotes para ciência de dados (pacotes de estatística, matemática, mineração de dados, inteligência artificial, aprendizado de máquina, cálculo numérico, etc.). Por isso, tem sido adotada por um número cada vez maior de empresas.
- Apesar do desenho das duas cobrinhas no logotipo, o nome “Python” não representa uma homenagem à cobra píton⁴. Na realidade, trata-se de uma referência⁵ ao grupo humorístico inglês Monty Python⁶.

³ Engenharia de software é a subárea da ciência computação que trata da pesquisa e desenvolvimento de técnicas que visem garantir alta qualidade e produtividade no processo de criação de softwares.

⁴ <https://www.youtube.com/watch?v=uns8vUQNxpc>

⁵ <https://docs.python.org/3/faq/general.html#why-is-it-called-python>

⁶ <https://www.imdb.com/title/tt0071853/>



Lição 2 – Distribuições do Python

Por ser uma “linguagem-tudo” (de propósito geral), gratuita e muito popular, Python possui um impressionante **ecossistema** de pacotes que já atingiu a casa das dezenas de milhares! Eles são destinados aos mais diferentes tipos de problemas: alguns servem para a criação de jogos, muitos deles são para o desenvolvimento de aplicações Web, alguns outros para o desenvolvimento de sistemas embarcados, há um bom número que oferece algoritmos de aprendizado de máquina, e por aí vai. Não há nem como tentar fazer uma lista completa, pois o número de categorias de problemas distintos cobertos pelos pacotes Python é realmente enorme!

Esta situação motivou o surgimento de diferentes **distribuições** do ambiente Python. Uma distribuição (ou *distro*), nada mais é do que uma coleção de pacotes e aplicativos selecionados, que são reunidos em um único **arquivo instalador** por serem considerados os “mais adequados” para uma determinada finalidade. A Figura 1 apresenta dois exemplos de distribuições muito conhecidas, CPython e WinPython.

Distribuição CPython	Distribuição WinPython
<ul style="list-style-type: none">• Interpretador Python• Standard Library• Utilitários básicos (IPython, pip, ...)	<ul style="list-style-type: none">• Interpretador Python• Standard Library• Utilitários básicos (IPython, pip, ...)• 'NumPy'• 'pandas'• 'Matplotlib'• 'scikit-learn'• 'SciPy'• 'Spyder'• 'Jupyter Notebook'• ...

Figura 1. Duas diferentes distribuições do ambiente Python: CPython e WinPython

CPython é a **distribuição oficial** do Python, gerenciada pela *Python Software Foundation*, cujo arquivo instalador para diferentes plataformas – Windows, Mac, Linux – pode ser obtido a partir de <https://www.python.org/>. Trata-se da versão de referência do Python, que vem basicamente com a dupla interpretador + *standard library* e um pequeno conjunto de utilitários básicos. Pode-se dizer que é uma distribuição “neutra”. Nenhum pacote específico para ciência de dados é instalado por esta distribuição. No entanto, o CPython disponibiliza um aplicativo chamado “**pip**” que possibilita a posterior instalação de qualquer pacote que faça parte do repositório PyPI. Informações sobre o CPython e o “pip” são apresentadas no Anexo B.

Por sua vez, a distribuição **WinPython** foi criada exclusivamente para o ambiente Windows, sendo destinada aos que desejam trabalhar com Python para ciência de dados. A sua característica mais atraente é fato de que ela não precisa ser instalada; na verdade, basta descompactar o arquivo instalador em alguma pasta de seu computador e, pronto, você já pode começar programar! A WinPython já vem com mais de 300 pacotes e aplicativos para ciência de dados: ‘NumPy’, ‘pandas’, ‘Matplotlib’, ‘Spyder’, etc., que podem ser usados imediatamente, sem a necessidade de nenhuma etapa de instalação adicional.

Devido à simplicidade de seu processo de configuração e pelo fato de ser uma distribuição voltada para ciência de dados, decidimos adotar a WinPython como o ambiente Python padrão para a execução dos exemplos apresentados neste livro. Na próxima lição, são apresentados os passos necessários para efetuar o seu download e instalação. No entanto, é importante observar que esta não é a melhor distribuição do Python, ela é apenas um bom ponto de partida para quem está começando a trabalhar com esta linguagem. Outro problema da WinPython é que ela não está disponível para Linux e Mac. Caso você utilize estes sistemas operacionais, uma alternativa interessante é utilizar a distribuição Anaconda (<https://www.anaconda.com/download/>), que também é voltada para ciência de dados, sendo inclusive bem mais abrangente (vem como mais de 700 pacotes). Outra opção é montar um ambiente a partir da distribuição CPython, porém instalando também os principais pacotes para ciência de dados com o uso do “pip”. No Anexo B, são fornecidas instruções para a instalação do CPython e sobre a forma de utilização do “pip”.

Versões do Python – Python 2 versus Python 3

- As versões do Python são numeradas como x.y.z (ex.: Python 3.8.1). Neste esquema de numeração, tem-se que:
 - “x” representa o número principal da versão, incrementado apenas quando ocorrem mudanças extremamente significativas na linguagem.
 - “y” é o número secundário, incrementado quando ocorrem mudanças menos significativas.
 - “z” é o nível micro, normalmente incrementado quando ocorrem *bugfixes* (correções de problemas).
- No ano de 2008, a *Python Software Foundation* decidiu promover uma mudança radical na linguagem, lançando a série de versões Python 3. Esta nova série não é completamente compatível com a série Python 2, o que significa que muitos programas implementados no Python versão 2.a.b podem não rodar no Python versão 3.c.d.
- Sendo assim, quando você baixar uma distribuição, deverá sempre optar pelo instalador referente ao Python 3. Não vale mais a pena trabalhar com o Python 2, visto que esta versão é considerada obsoleta e deverá ser descontinuada ao longo de 2020 (ao menos, esta é a previsão dada pela *Python Software Foundation*).



Lição 3 – Instalação do WinPython

A seguir são apresentados os passos para a instalação do WinPython em uma máquina com sistema operacional Windows 7 ou superior.

3.1 Instalação em sistemas Windows de 64 bits:

- **3.1.1 Acessar o repositório do WinPython:** <https://sourceforge.net/projects/winpython/> (Figura 2).

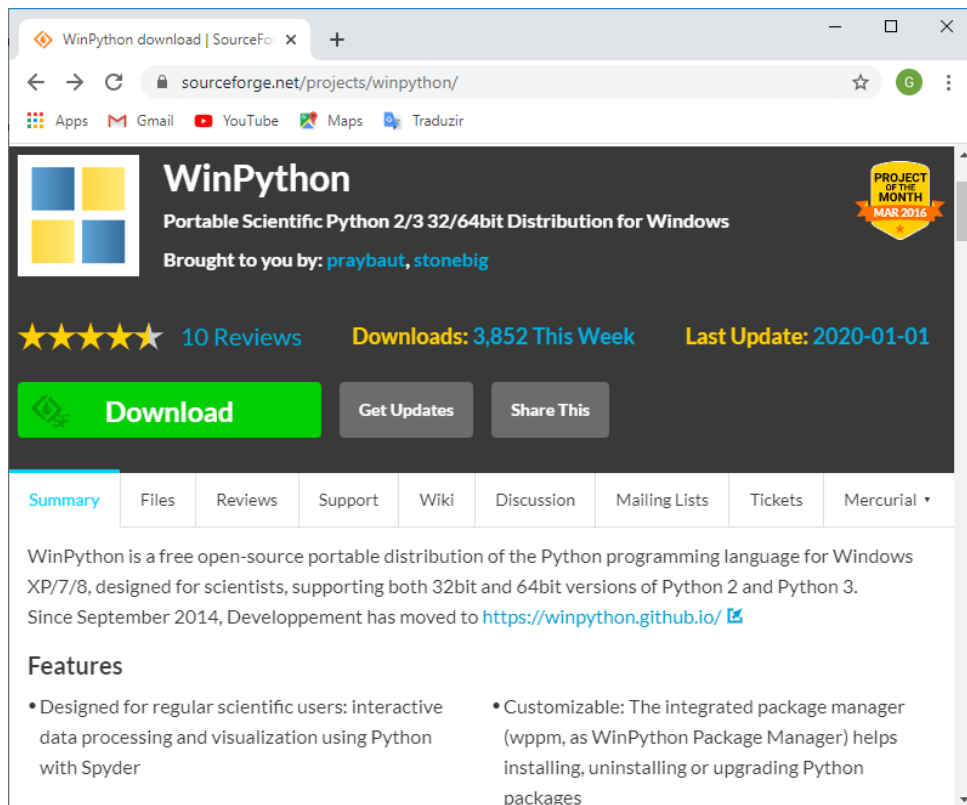


Figura 2. Web site para download do WinPython

- **3.1.2 Efetuar o download do arquivo de instalação (versão 64 bits):** caso o seu sistema operacional seja de **64 bits**, clique no botão **Download**. Após, alguns segundos, o arquivo de instalação começará a ser automaticamente baixado. No momento da elaboração deste livro, a última **versão estável** do *software* era a de número 3.7.6.0 (por isso, o arquivo tinha o nome de “WinPython64-3.7.6.0.exe”). O processo de *download* levará algum tempo até ser concluído, pois o arquivo de instalação possui mais de 648MB.
- **3.1.3 Instalar o WinPython:** ao término do *download*, execute o arquivo baixado para iniciar o processo de instalação do WinPython em sua máquina (que na verdade consiste em uma simples descompactação). A instalação é demorada, mas extremamente simples, consistindo apenas em indicar o local em que a pasta do WinPython (denominada “WPy64-3760”) será descompactada e então clicar no botão Extract. Na instalação que realizei em minha máquina, optei por trocar a sugestão inicial (pasta Downloads) para um caminho mais simples (“C:\”), pois assim o WinPython pôde ser descompactado diretamente para a pasta C:\WPy64-3760. Com relação ao espaço ocupado em disco, o WinPython é um pouco guloso, exigindo mais de 3GB de espaço livre no seu HD.

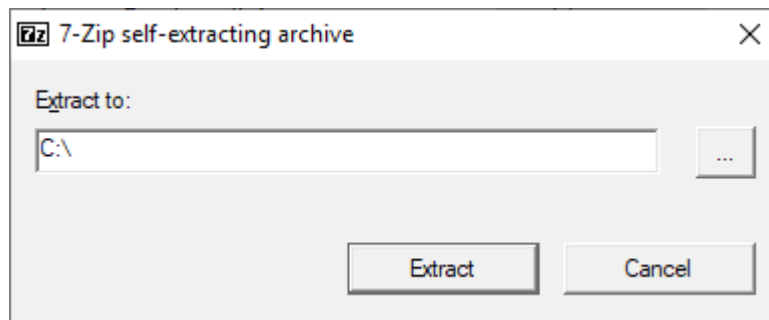


Figura 3. Instalação do WinPython: modificando a pasta destino

3.2 Instalação em sistemas Windows de 32 bits:

- **3.2.1 Navegar até a pasta “Files”:** a instalação do WinPython em sistemas de 32 bits começa um pouquinho diferente, pois, antes de tudo, é preciso navegar até a página que contém as diferentes versões da distribuição. Sendo assim, depois de acessar o repositório do WinPython (<https://sourceforge.net/projects/winpython/>), você deverá clicar em **Files**, conforme destacado na Figura 4 (não é para clicar no botão Download, mas sim em Files!!!).

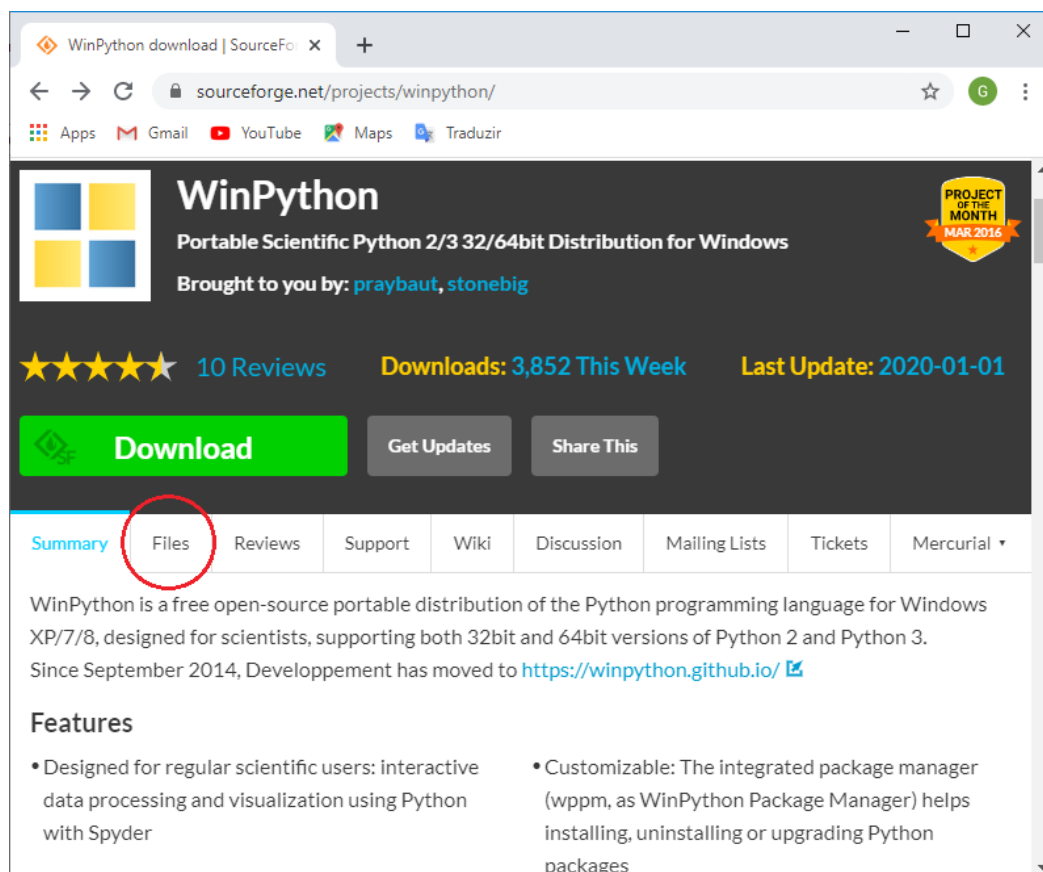


Figura 4. Download do instalador do WinPython de 32 bits (parte 1)

- **3.2.2 Efetuar o download do arquivo de instalação (versão 32 bits):** aparecerá a tela da Figura 5, onde as diversas versões da distribuição são listadas, cada qual com o seu link para *download*. Observe que há um botão verde informando que a última **versão estável** é

“WinPython64-3.7.6.0” (o botão em que está escrito “Download Latest Version”). **Não** clique nesse botão, pois ele acarretará no download da versão de 64 bits. Em vez disso, clique no link **WinPython_3.7** (destacado na Figura 5) para que seja possível acessar a versão 32 bits do WinPython 3.7.6.0. Na janela que será aberta, clique então em **3.7.6.0**. Por fim, na nova página que abriu, role a tela para baixo e clique no arquivo “WinPython32-3.7.6.0.exe”. Pronto! O arquivo começará a ser baixado.

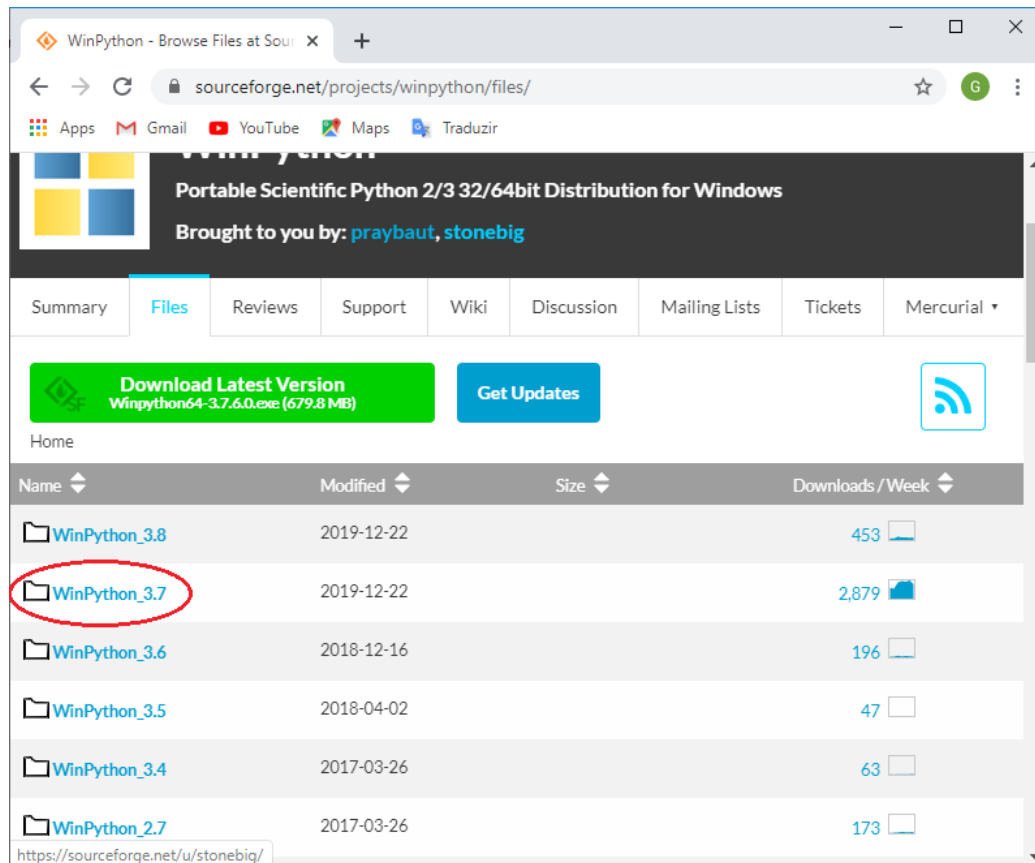


Figura 5. Download do instalador do WinPython de 32 bits (parte 2)

- **Importante:** o grande macete na instalação de 32 bits é primeiro identificar qual é o número da última versão estável (isto é feito observando-se o botão verde, onde está escrito “Download Latest Version”). Feita a identificação, você deverá clicar no link que possui o número da versão para poder ter acesso ao instalador de 32 bits.
- **3.2.3 Instalar o WinPython:** Terminado o *download*, basta seguir as orientações apresentadas no item 3.1.3 para instalar o WinPython.



Lição 4 – Utilizando o WinPython no Modo Interativo

Os programas Python podem ser executados em dois modos: **script** (ou normal) e **interativo**. No modo script, o programa é primeiro digitado por completo, em seguida salvo e, por fim, executado “de cabo a rabo” (isto é, por inteiro, sem pausas) pelo interpretador Python. De maneira

oposta, quando estamos trabalhando no modo interativo, o Python interpreta e executa comando por comando, conforme eles vão sendo digitados. A utilização do WinPython no modo normal será tratada na próxima lição. Nesta, apresentaremos a receita para trabalhar com o WinPython no modo interativo. Basta seguir os passos abaixo:

- **4.1 Iniciar o QtConsole:** abra o Windows Explorer e acesse a pasta que você escolheu para descompactar o WinPython. Então, efetue o duplo clique sobre o programa “**IPython Qt Console.exe**” (Figura 6). Trata-se de uma versão um pouco mais “simpática” do IPython, o mais conhecido terminal interativo do Python.

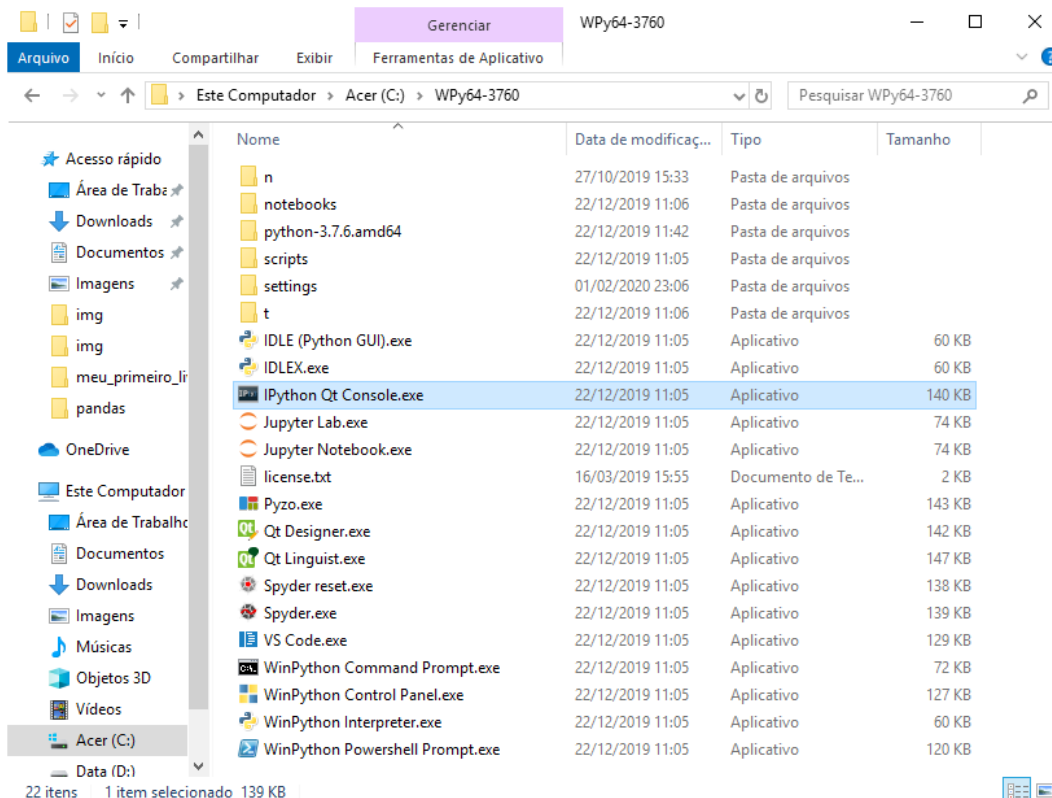


Figura 6. Localizando o aplicativo “QtConsole”

- Uma tela similar à mostrada na Figura 7 será aberta.

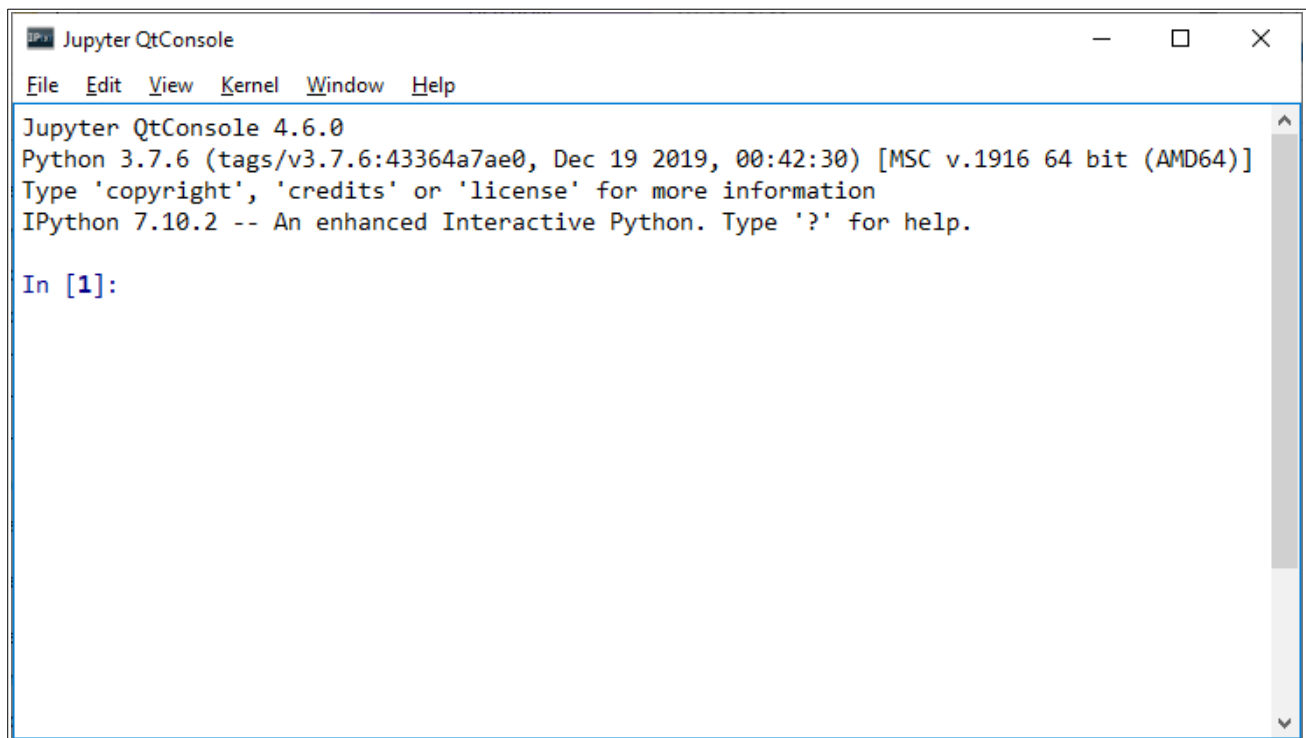


Figura 7. Tela inicial do QtConsole

- **4.2 Digitando o seu primeiro comando:** podemos testar o WinPython digitando simplesmente $1 + 1$. Você vai digitar na parte da tela que exibe “In [1]:” (chamada de *prompt* de entrada). Basta fazer como indicado abaixo:

```
In [1]: 1 + 1
Out[1]: 2

In [2]:
```

- O valor 2 será retornado após a indicação “Out[1]:” e, logo abaixo, vai aparecer um novo *prompt* para entrada de dados, desta vez representado por “In [2]:”. O modo interativo funciona sempre desta maneira: digitamos uma linha de comando e o Python a processa imediatamente, ficando logo em seguida pronto para que possamos digitar um outro comando.
- **4.3 Digitando mais um comando:** dessa vez, podemos arriscar uma coisa mais clássica: o comando para escrever “Olá Python!” na tela. Basta utilizar a função `print()` e escrever a mensagem entre aspas simples, dentro de parênteses:

```
In [1]: 1 + 1
Out [1]: 2

In [2]: print('Olá Python!')
Olá Python!

In [3]:
```

- **4.4 Encerrando a conversa com o Python:** quando quiser fechar o QtConsole digite `quit()` ou simplesmente feche a janela do aplicativo.

```
In [1]: 1 + 1
Out [1]: 2

In [2]: print('Olá Python!')
Olá Python!

In [3]: quit()
```



Lição 5 – Utilizando o WinPython no Modo Script

Ambiente integrado de desenvolvimento (*Integrated Development Environment* – IDE) é um jargão utilizado na área de engenharia de software para rotular aplicativos destinados a servir como ambientes para a criação, depuração e execução de programas em uma determinada linguagem. A grande vantagem de utilizar uma IDE é que normalmente este tipo de software oferece uma série de recursos capazes de aumentar a eficiência do processo de programação. Alguns exemplos: **autocomplete** (possibilita com que um comando seja automaticamente completado durante a digitação), **inline help** (exibe um texto de ajuda sobre uma palavra reservada quando o mouse é colocado sobre a mesma) e **formatação automática** de programas (realiza o alinhamento automático de comandos que estejam colocados entre o início e o fim de um bloco de instruções). Apenas para apresentar alguns exemplos bem conhecidos de IDEs, podemos citar o aplicativo Eclipse, para a criação de programas na linguagem Java, e o RStudio, para o desenvolvimento em R.

Existem muitas e muitas IDEs disponíveis para Python, como PyCharm, Rodeo, Jupyter Notebook e Spyder. Na Internet, você encontrará bons artigos que discutem as vantagens e desvantagens de cada uma delas. Neste livro, adotaremos a **Spyder**, uma vez que esta IDE é otimizada para projetos de ciência de dados, além de ser muito intuitiva e permitir o uso do Python em ambos os modos, normal e interativo. Para quem está acostumado a programar em R, eu devo admitir que a Spyder não é tão bacana quanto o RStudio, além de ser um software um pouco pesado, que exige uma quantidade considerável de memória para ser executado. Mas, de qualquer forma, é bem mais agradável do que o QtConsole. O exemplo a seguir mostra o passo a passo para você utilizar o Python no **modo script** por meio da Spyder.

- **5.1 Iniciar a IDE Spyder:** abra o Windows Explorer e acesse a pasta que você escolheu para descompactar o WinPython. Em seguida, efetue o duplo-clique sobre o programa “Spyder.exe” (Figura 8). Após algum tempinho, a tela principal da Spyder será mostrada (Figura 9).

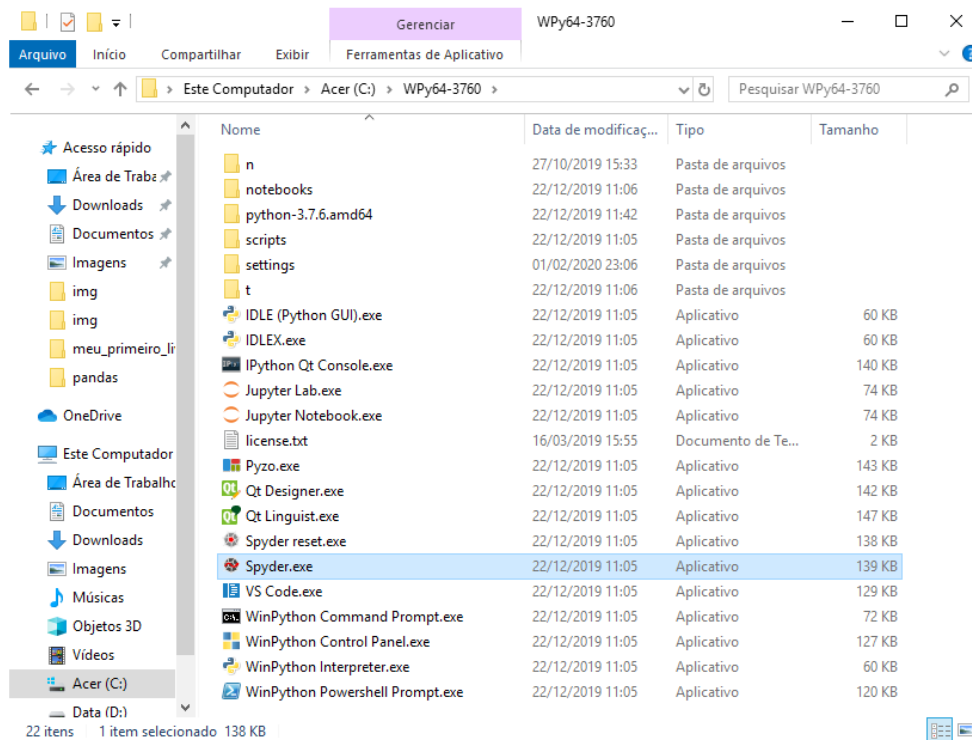


Figura 8. Localizando o aplicativo "Spyder"

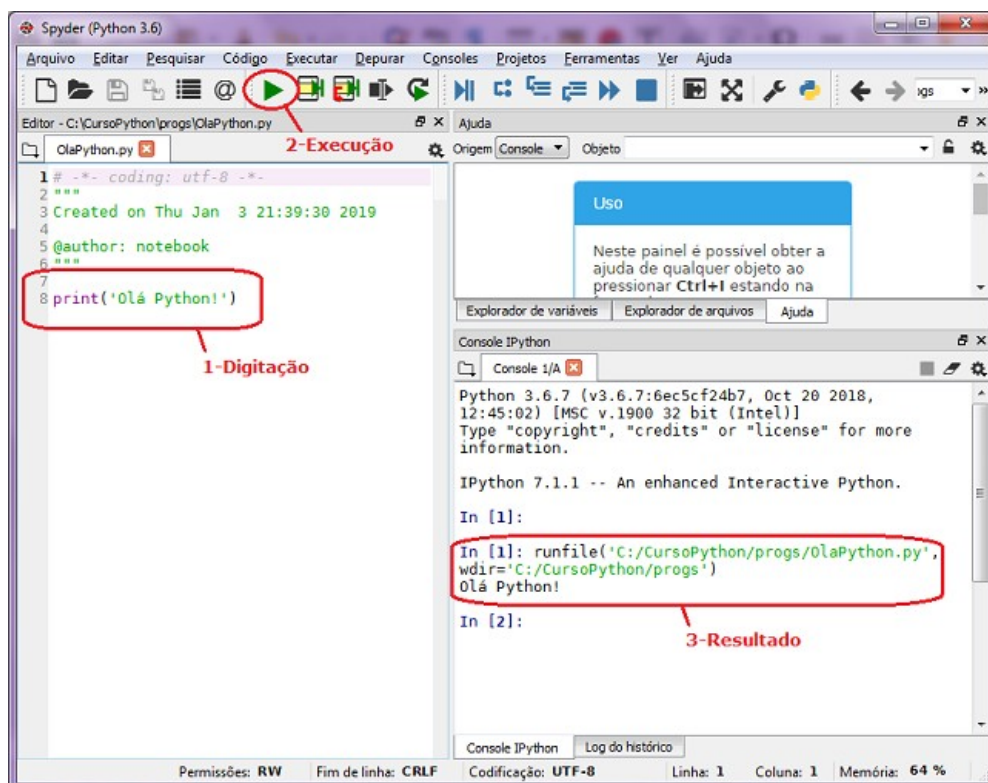


Figura 9. Tela principal do aplicativo "Spyder". Possui três áreas principais: (1)-Editor de Código, onde os programas são digitados; (2) – Menu, com destaque para o botão "Run"; (3) – Console IPython, onde as saídas são exibidas

- **5.2 Digitar o programa:** digite na área do editor de código um programa com uma única linha: `print('Olá Python!')`. Não se assuste com o fato de a Spyder já colocar de forma automática algumas linhas de cabeçalho (linhas compreendidas entre `#-*- coding: utf-8 -*-` e `"""`). Basta digitar o seu programa logo abaixo das mesmas, a partir da linha 8.
- **5.3 Salvar o programa:** o programa está digitado, mas ainda não foi salvo. Há duas formas de fazer isto: com o uso do menu **Arquivo > Salvar** ou digitando **CTRL + S**. Em seguida, escolha uma pasta e dê um nome para o programa. No exemplo mostrado na Figura 9, a pasta escolhida foi “C:\CursoPython” e o nome do programa “OlaPython.py”. A extensão “.py” foi utilizada porque é a extensão padrão de programas Python (assim como “.java” é usada por programas Java, “.c” por programas C, etc.).
- **5.4 Executar o programa:** existem algumas formas distintas:
 - Selecione a opção de menu **Executar > Executar**; ou
 - Clique no botão “executar arquivo” (destacado na Figura 9);
 - Simplesmente pressione F5.
- **5.5 Verificar os resultados:** a saída do programa (resultado da execução) será apresentada no **Console IPython**, que consiste na área localizada no canto inferior direito da tela (indicada por “3-Resultados” na Figura 9). Este console também permite com que você utilize o Python no modo interativo.

Agora que você já instalou o WinPython e sabe como criar e executar programas tanto no modo interativo como no normal, respire fundo e se prepare! As próximas lições deste livro foram cuidadosamente preparadas para que você se transforme em um verdadeiro *pythonista*!



Lição 6 – Programação Python: Primeiros Passos

6.1 Variáveis e Tipos

Uma variável pode ser entendida como o nome de um local onde se pode colocar qualquer valor numérico ou não-numérico. O código a seguir exemplifica o uso de variáveis em programas Python e apresenta os tipos de dado básicos oferecidos pela linguagem. Neste exemplo tanto como nos demais apresentados ao longo no livro, mostra-se primeiro o código do programa e, logo abaixo, o resultado da execução. O resultado estará sempre listado após o prompt “Saída [n]:”, onde *n* é o número do programa.

Programa 1 – Utilização de variáveis para armazenar informações de uma pessoa.

```
#P001: variáveis e tipos; funções type() e print()

#PARTE 1: declaração de variáveis
nome = 'Jane'
sobrenome = "Austen"
idade = 41
nota = 9.9
aprovado = True

#PARTE 2: imprimindo o conteúdo das variáveis e os tipos das mesmas
print(nome, sobrenome, idade, nota, aprovado)
print(type(nome))
print(type(sobrenome))
print(type(idade))
print(type(nota))
print(type(aprovado))

#PARTE 3: mudando o valor e o tipo da variável "nota"
nota = 'A'
print('mudei o valor e o tipo de "nota" para: ', nota, ",", type(nota))
```

Saída [1]:

```
Jane Austen 41 9.9 True
<class 'str'>
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
mudei o valor e o tipo de "nota" para: A , <class 'str'>
```

Para começar, é importante mencionar que este é um típico exemplo de programa procedural, cujo **fluxo de execução** (ordem em que os comandos são executados pelo interpretador Python) é de cima para baixo, em sequência. O programa foi dividido em três partes. Vamos às explicações sobre cada uma:

- Na PARTE 1, encontram-se as declarações das variáveis “nome”, “sobrenome”, “idade”, “nota” e “aprovado”. Estas variáveis são declaradas, respectivamente, com valores do tipo **str** (abreviação de **string**, isto é, alfanumérica, tanto “nome” quanto “sobrenome”), **int** (valores inteiros), **float** (valores reais) e **bool** (valores lógicos: `True` ou `False`). Estes são os quatro **tipos de dados primitivos** do Python. O nome “primitivo” refere-se ao fato de que estes são os tipos de valores mais simples com os quais a linguagem trabalha.
 - O **operador de atribuição**, representado pelo símbolo “=”, é usado para atribuir um valor (lado direito da equação) a uma variável (lado esquerdo). Por exemplo: `nota = 9.9`. O valor associado não precisa ser sempre fixo; ele pode também ser outra variável (por exemplo, `a = b`) ou uma expressão (`perimetro_do_quadrado = 4 * lado`).
 - Em Python, a declaração de variáveis pode ser realizada em qualquer linha do programa. O tipo da variável será automaticamente determinado de acordo com o valor que lhe for atribuído, podendo mudar durante a execução do programa caso o valor da variável seja alterado.

- Os valores de strings podem ser definidos entre aspas simples (nome = 'Jane') ou aspas duplas (sobrenome = "Austen").
- A PARTE 2 cobre o uso de **print()**, a função padrão para saída de dados. Para imprimir diversas variáveis, basta separá-las por vírgula. Observe que valores de qualquer tipo podem ser impressos com esse comando. Ainda nesta parte, utilizamos a função **type()**, uma função muito útil que retorna o tipo de dado armazenado em uma variável.
- Na PARTE 3, mostra-se como uma variável pode ter não apenas o seu valor, mas também seu tipo modificado durante o fluxo de execução. Neste trecho de código, o valor de “nota” é mudado de 9.9 (tipo float) para ‘A’ (tipo str).

6.2 Comentários

Comentários são textos ou frases definidos com o uso do símbolo **#**. No programa anterior, observe que a primeira linha contém um texto com comentários:

```
#P001: variáveis e tipos; funções type() e print()
```

Os comentários são **ignorados** pelo interpretador Python na hora em que o programa é processado. Eles servem apenas para **documentação**, isto é, para serem lidos pelos humanos que estiverem analisando ou alterando o programa. Na prática, são muito úteis para que o entendimento de uma **rotina** (conjunto de instruções) complexa possa ser facilitado.

6.3 Expressões

O programa a seguir cobre as operações matemáticas básicas. Os operadores +, -, *, /, e ** são usados, respectivamente, para realizar a adição, subtração, multiplicação, divisão e exponenciação. O operador // retorna o quociente de uma divisão inteira, enquanto % retorna o módulo (resto). Qualquer operação segue a ordem básica conhecida como PEMDAS (parênteses, exponenciação, multiplicação, divisão, adição e subtração).

Programa 2 – Operações matemáticas básicas.

```
#P002: matemática básica

#Adição, subtração, multiplicação e divisão
x=5; y=2
print(x+y, x-y, x*y, x/y)
print(y**x)

#quociente e resto (ou módulo)
quociente = x // y
modulo = x % y
print("O quociente da divisão de", x, "por", y, "é: ", quociente)
print("O módulo da divisão de", x, "por", y, "é: ", modulo)

#expressão com parênteses
minha_expressao = (1 + 2) * 5**2 / ((5-3) + 1)
print("O valor da expressão 'minha_expressao' é: ", minha_expressao)

#a divisão de dois inteiros sempre gera um float
```

```
# (mesmo que a divisão seja exata)
a=10; b=5; c=a/b
print(a, b, c)
print(type(a), type(b), type(c))
```

Saída [2]:

```
7 3 10 2.5
32
O quociente da divisão de 5 por 2 é: 2
O módulo da divisão de 5 por 2 é: 1
O valor da expressão 'minha_expressao' é: 25.0
10 5 2.0
<class 'int'> <class 'int'> <class 'float'>
```

Duas observações importantes:

- É possível escrever mais de um comando em uma única linha, bastando separá-los por ponto e vírgula. Por exemplo: `a=10; b=5; c=a/b`. Entretanto, de acordo com as “boas práticas de programação” (PEP-8), você deve evitar produzir uma linha que possua mais de 72 caracteres.
- Conforme mostrado nas três últimas linhas do programa, no Python 3, a divisão de dois inteiros sempre resultará em um valor do tipo real (float).

6.4 Entrada de Dados via Teclado

A linguagem Python possui uma função chamada `input()` que possibilita a entrada de dados via teclado.

Programa 3 – Recebe o nome de uma pessoa via teclado.

```
#P003: Entrada de dados
nome = input("Qual o seu nome?\n")
print("Hmmm... então você é o famoso " + nome)
```

Saída [3]:

```
Qual é o seu nome?
> Nelson Cavaquinho
Hmmm... então você é o famoso Nelson Cavaquinho
```

Explicando:

- Quando a função `input()` é chamada, o programa “dá uma paradinha” e fica esperando o usuário digitar alguma coisa. Após o usuário terminar de digitar e teclar Enter, a função retornará o valor introduzido via teclado como uma string. Este valor poderá então ser armazenado em uma variável. Em nosso exemplo, foi armazenado na variável “nome”.
- Na primeira linha do programa, a estranha sequência “`\n`” foi utilizada para forçar uma quebra de linha após a pergunta “Qual é o seu nome?”. Essa sequência também é comumente utilizada dentro da função `print()`. Na realidade, ela é muito empregada em programas escritos em diferentes linguagens de programação, sendo intitulada “formato *newline*”.

- Na segunda linha do programa, o operador `+` é empregado para concatenar strings, ou seja, para juntar a frase “Hmm... então você é o famoso” com o valor da variável “nome”.
- Na ciência de dados, os programas costumam receber como entrada apenas arquivos ou bases de dados e, por isso, a função `input()` praticamente não é utilizada.

Nomes de Variáveis

- Como foi introduzido na Lição 1, Python é uma linguagem que prioriza a legibilidade do código. Por este motivo, os *pythonistas* são incentivados a escolher nomes que consigam expressar com clareza o significado das variáveis (mesmo que eles fiquem grandes).
- Existe até mesmo um guia de estilo denominado PEP-8, responsável por oferecer uma série de regras que o programador pode aplicar para deixar o seu código mais legível⁷. Este guia recomenda que todas as variáveis devam ter o nome especificado em minúsculo e que o caractere *underscore* (`_`) seja utilizado para separar nomes compostos, padrão este conhecido como *snake case*. Alguns exemplos:
 - `idade`
 - `renda_media_anual`
 - `codigo_ocupacao`
 - `produtos_2019`
- Os nomes podem conter letras, números e o caractere *underscore*, no entanto não podem ser iniciados por um número. Com relação ao *underscore*, nós até podemos começar nomes com ele, mas apenas em situações específicas⁸ – portanto, evite fazer isso por enquanto! (maiores informações sobre esse assunto serão dadas no Capítulo IX).
- Há diferenciação entre letras maiúsculas e minúsculas, o que significa que se você nomear uma variável como “x”, **não** poderá referenciá-la como “X” (mas lembre-se de que a recomendação é usar minúsculo nos nomes de todas as variáveis).
- O nome de uma variável também não pode ser igual ao de uma **palavra reservada** (*keyword*) da linguagem Python, cuja lista completa contendo 33 nomes é apresentada abaixo:

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

⁷ PEP é o acrônimo para *Python Enhancement Proposals* ou Propostas para Melhorias no Python. Além da PEP-8 que trata especificamente do guia de estilo, existem várias outras PEPs cobrindo os mais diferentes temas.

⁸ <https://www.datacamp.com/community/tutorials/role-underscore-python>



Lição 7 – Instruções de Desvio Condicional: if, else, elif

Em qualquer *script* de análise de dados, sempre ocorrerão situações em que será necessário analisar condições para, só então, definir o comportamento do programa. As instruções **if** e **else** são as utilizadas para esta finalidade. Elas são chamadas de instruções de **desvio condicional**.

Programa 4 – Desvio condicional com if: - else:. Neste exemplo, a estrutura **if-else** é empregada para verificar a idade de uma pessoa que deseja entrar em uma festa realizando uma comparação com o valor inteiro 18 (a idade da pessoa está armazenada na variável “idade”). Se (**if**) a idade for maior ou igual a 18, então uma mensagem que a convida a entrar é exibida na tela. Senão (**else**), uma mensagem informando que o cidadão ou cidadã está devidamente barrado(a) é exibida na tela (não adianta nem a pessoa argumentar que tem 17 anos e 11 meses, pois “idade” é do tipo inteiro!!!). Se você tiver alguma dúvida, teste o programa diferentes vezes modificando o valor da variável “idade”.

```
#P004: desvio condicional com if: - else:
idade = 17
if (idade >= 18) :
    print("Pode entrar, a festa está bombando!")
    print("Temos muita música e drinks especiais!!!")
else :
    print("Você é jovem demais para este clube! Volte apenas quando fizer 18.")
```

Saída [4]:

Você é jovem demais para este clube! Volte apenas quando fizer 18.

Observe que os comandos **if** e **else** terminam com um sinal de dois pontos “:” e que as **linhas subordinadas** a cada comando estão **indentadas**. Neste exemplo, o **if** tem dois comandos subordinados e o **else** apenas um. No jargão da computação, um conjunto de comandos subordinados é chamado de **bloco de código**. A seção a seguir descreve em detalhes o esquema utilizado pelo Python para a definição de blocos de código.

7.1 Blocos de Código

Para começar, um aviso: preste atenção, muita atenção mesmo, pois será apresentada uma característica muito peculiar do Python. Ao contrário do que ocorre em qualquer outra linguagem de programação, os blocos de código que estão subordinados ao **if**, **else** e a outros comandos que ainda serão apresentados, como **for** e **while**, são definidos através da **indentação de comandos**⁹. Enquanto o Pascal usa as palavras *begin* e *end* para marcar um bloco e linguagens como C, Java e R utilizam os símbolos “{” e “}”, o Python utiliza espaços em branco ou tabulações. É isso mesmo: espaços em branco ou tabulações. Isto significa que os comandos precisam estar alinhados da mesma forma para que o Python os reconheça como parte de um bloco. A convenção é utilizar 4 espaços para indentação¹⁰ (esta é mais uma recomendação da PEP-8). Entretanto, a linguagem permite qualquer padrão de alinhamento (por exemplo, um tab, três espaços, etc.), contanto que este seja repetido para todos os comandos pertencentes a um mesmo bloco. Para que o conceito fique claro, observe o exemplo seguinte:

⁹ A palavra “indentação” é muito feia, mas infelizmente existe!

¹⁰ <https://stackoverflow.blog/2017/06/15/developers-use-spaces-make-money-use-tabs/>

Erro de indentação: <pre>if (x > 0) : a=1</pre> <p><code>IndentationError: expected an indented block</code></p>	Indentação correta: <pre>if (x > 0) : a=1</pre>
---	--

Cuidado com o Copiar e Colar!

- Se você tentar selecionar e copiar o código dos programas a partir deste livro em PDF, perceberá que eles perderão a indentação ao serem colados em seu destino. No entanto, não se esqueça de que este livro vem acompanhado de um arquivo ZIP ("CursoPython.zip") que contém todos os programas e bases de dados apresentados ao longo dos capítulos (qualquer dúvida, leia a Seção **Propósito**). Basta então copiar e colar o código a partir dos programas do arquivo ZIP, pois assim a indentação será preservada.

7.2 Operadores Relacionais e Lógicos

A instrução **if** toma uma decisão a partir da avaliação de uma **condição**. Uma condição representa uma comparação ou um conjunto de comparações que irá resultar sempre em um valor do tipo lógico (boolean), isto é, VERDADEIRO (True) ou FALSO (False). Na linguagem Python, os operadores utilizados nas condições – denominados **operadores relacionais** – são os seguintes:

- `x == y` *# o valor de x é igual ao de y?*
- `x != y` *# o valor de x é diferente do valor de y?*
- `x > y` *# o valor de x é maior que o de y?*
- `x < y` *# o valor de x é menor que o de y?*
- `x >= y` *# o valor de x é maior ou igual ao de y?*
- `x <= y` *# o valor de x é menor ou igual ao de y?*
- `x is y` *# x e y apontam/não apontam para o mesmo endereço de memória? (detalhes no Capítulo III)*
- `x is not y`

Os operadores lógicos **and** (e), **or** (ou) e **not** (não) podem ser utilizados para unir diferentes condições, aumentando assim, a complexidade dos testes. Eles funcionam da seguinte maneira:

- `and` *# a sentença é verdadeira se TODAS as condições forem verdadeiras*
- `or` *# a sentença é verdadeira se UMA das condições for verdadeira*
- `not` *# inverte o valor lógico de uma sentença
(True → False, False → True)*

Programa 5 – Miscelânea de exemplos envolvendo operadores relacionais e lógicos.

```
#P005: operadores relacionais e lógicos
print('* * * * parte 1 * * * * ')
print(4 * 2 == 8)           #True.
print(9 ** 2 == 81)         #True.
print(5 + 2 < 7)            #False.
print(5 + 2 >= 7)           #True.
print('Portela' == 'Mangueira') #False.
print(5 / 2 > 3)            #False.
print(7 % 2 != 0)           #True.

print('\n\n* * * * parte 2 * * * * ')
pais = 'Brasil'
print(pais == 'Brasil') #True.
print(pais == 'BRASIL') #False.

media_final = 7.0
if (media_final >= 7.0):
    print('com a média', media_final, 'você está aprovado')
else:
    print('reprovado')

a=10; b=100
if (a >= b):
    print('o valor de "a" é maior ou igual ao de "b"')
else:
    print('o valor de "a" é menor do que o de "b"')

print('\n\n* * * * parte 3 * * * * ')
print((4 * 2 == 8) and (9**2 >= 81)) #True.
print((1 + 1 < 3) and (9**2 > 81))  #False.
print((1 + 1 < 3) or ('cruzeiro' == 'atletico')) #True.
```

Saída [5]:

* * * * parte 1 * * * *

True
True
False
True
False
False
True

* * * * parte 2 * * * *

True
False
com a média 7.0 você está aprovado
o valor de "a" é menor do que o de "b"

* * * * parte 3 * * * *

True
False
True

Em situações onde mais de dois operadores lógicos tenham que ser utilizados para a montagem de um teste lógico, a ordem de precedência apresentada no Quadro 1 será adotada.

Quadro 1. Regras de precedência para operadores lógicos

Operador	Ordem de avaliação
not	1
and	2
or	3

O operador **not** tem maior prioridade (é o primeiro a ser avaliado), seguido do **and** e do **or**, respectivamente. Estas regras de precedência podem ser sobrepostas através do uso de parênteses. Para que o conceito fique claro, considere o cenário descrito a seguir. Suponha que você está desenvolvendo um programa para processar uma base de dados de produtos e que, nesta base, existam cinco diferentes categorias de produtos: 'A', 'B', 'C', 'D' e 'E'. Imagine que você deseja produzir um relatório que listará apenas os produtos das categorias 'A' e 'B' que custem menos de R\$ 500,00. Em uma situação como esta, caso você monte a sua estrutura de desvio da forma apresentada abaixo, acabará por selecionar produtos que não deveriam pertencer à listagem:

```
preco = 999.99; categoria = 'B'

if categoria=='B' or categoria=='A' and preco < 500:
    print('selecionado')
else:
    print('não selecionado')
```

Saída [1]:
selecionado

Neste exemplo, temos um produto da categoria 'B' com preço igual a R\$ 999,99, ou seja, acima de R\$ 500,00. Ele não deveria ser selecionado, mas a instrução **if** possui um erro de lógica: como o operador **and** tem precedência sobre o operador **or**, o Python testa a condição especificada em negrito primeiro. Ou seja, o teste será interpretado pelo Python da seguinte maneira: “selecione o produto caso a categoria seja igual a 'A' e o preço for menor do que 500; ou se a categoria for 'B'”. Observe agora a correção do programa, onde os parênteses são empregados para alterar a precedência do teste e, desta forma, pegar apenas os produtos corretos:

```
preco = 999.99; categoria = 'B'

if (categoria=='B' or categoria=='A') and preco < 500:
    print('selecionado')
else:
    print('não selecionado')
```

Saída [1]:
não selecionado

7.3 Instrução **elif**

Em muitas situações práticas, existe a necessidade de avaliar mais de duas possibilidades em um teste condicional. Nesta situação, os testes poderão ser logicamente agrupados com o uso da

instrução **elif** (não é “else if” e tampouco “elsif”, o nome da instrução é “elif”). No Programa 6, a estrutura if-elif-else é utilizada para determinar a categoria de um valor numérico.

Programa 6 – Retorna a faixa etária de uma pessoa em função de sua idade.

```
#P006: if, elif, else
idade = 25

if (idade < 18):
    faixa_etaria = '<18'
elif (idade >= 18 and idade < 30):
    faixa_etaria = '18-29'
elif (idade >= 30 and idade < 40):
    faixa_etaria = '30-39'
else:
    faixa_etaria = '>=40'

print('Se a idade é', idade, 'então a faixa etária é :', faixa_etaria)
```

Saída [6]:

Se a idade é 25 então a faixa etária é : 18-29

Python **não** possui uma estrutura do tipo case ou switch presente em outras linguagens de programação. Ou seja: o que você implementaria com case-switch em outra linguagem, você fará com if-elif-else em Python.

7.4 Instruções Condicionais Aninhadas (*Nested Conditionals*)

Para encerrar a lição, a seguir apresentamos um exemplo de código que contém um if dentro de um else. Para implementar uma rotina deste tipo – if dentro de else, if dentro de if, ou qualquer coisa parecida – também é preciso fazer uso da indentação.

Programa 7 – Dados dois números, verifica qual é o maior ou se ambos são iguais.

```
#P007: instruções condicionais aninhadas
a=5; b=10
if (a == b):
    print("a e b são iguais")
else:
    if (a > b):
        print("a é maior do que b")
    else:
        print("a é menor do que b")
```

Saída [7]:

a é menor do que b

O Quadro 2, na página a seguir, resume a sintaxe dos comandos de desvio condicional.

Quadro 2. Sintaxe da estruturas: (1) if (sozinho); (2) if – else; (3) if – elif – else;

- if sozinho (podendo ter um ou mais comandos subordinados)

```
if <condição> :  
    comando1  
    ...  
    comandon
```

- if com else (ambos podendo ter um ou mais comandos subordinados)

```
if <condição> :  
    comando1  
    ...  
    comandon  
else :  
    comando1  
    ...  
    comandon
```

- if + elif + else (cada qual podendo ter um ou mais comandos subordinados)

```
if <condição1> :  
    comando1  
    ...  
    comandon  
  
elif <condição2> :  
    comando1  
    ...  
    comandon  
...  
elif <condiçãom> :  
    comando1  
    ...  
    comandon  
else :  
    comando1  
    ...  
    comandon
```



Lição 8 – Instruções de Repetição (1): while

Uma estrutura de **repetição** (ou **laço**) permite com que um bloco de instruções possa ser executado diversas vezes até que uma condição seja satisfeita. Na linguagem Python, há dois tipos de estrutura de repetição: **while** e **for**.

O comando **while** funciona da mesma forma que em outras linguagens de programação: enquanto o valor da condição especificada ao lado da palavra **while** for verdadeira, o bloco de código subordinado ao comando é executado. Quando for falso, o comando é abandonado. Se no

primeiro teste o resultado é falso, os comandos não são executados nenhuma vez. A sintaxe do comando é apresentada no Quadro 3.

Quadro 3. Sintaxe do comando while

```
while <condição> :  
    comando1  
    ...  
    comandon
```

Programa 8 – Tabela de equivalência entre graus Celsius e graus Fahrenheit. No programa a seguir, gera-se uma tabela em que -20°C é o valor de temperatura inicial e 100°C o final. A escala da tabela em graus Celsius varia de 10 em 10.

```
#P008: repetição com o comando while (primeiro exemplo)  
c = -20  
print('* * * Tabela de conversão de graus Celsius para graus Fahrenheit')  
while c <= 100:  
    f=c*1.8 + 32  
    print(c, '°C ----> ', f, '°F')  
    c=c+10  
  
print('\nFIM!!!')
```

Saída [8]:

* * * Tabela de conversão de graus Celsius para graus Fahrenheit

```
-20 °C ----> -4.0 °F  
-10 °C ----> 14.0 °F  
0 °C ----> 32.0 °F  
10 °C ----> 50.0 °F  
20 °C ----> 68.0 °F  
30 °C ----> 86.0 °F  
40 °C ----> 104.0 °F  
50 °C ----> 122.0 °F  
60 °C ----> 140.0 °F  
70 °C ----> 158.0 °F  
80 °C ----> 176.0 °F  
90 °C ----> 194.0 °F  
100 °C ----> 212.0 °F
```

FIM!!!

A variável “c” armazena a temperatura em graus Celsius e também atua como **variável de controle**, pois ela é utilizada para controlar o laço, ou seja, para mantê-lo em execução durante o número de repetições desejadas. Ela é inicializada fora do laço e é atualizada dentro do mesmo, como deve ocorrer sempre que utilizarmos o **while**. Cada repetição executada dentro de um laço é chamada de **iteração**. Desta forma, o programa acima realizou 13 iterações, pois as três instruções subordinadas ao **while** foram executadas 13 vezes.

Programa 9 – Computa o valor da série $H = 1 + (1 / 2) + (1 / 3) + \dots + (1 / N)$.

```
#P009: repetição com o comando while (segundo exemplo)
N = 5
H = 1
i = 1 #variável de controle

print('* * * cálculo de H = 1 + (1 / 2) + (1 / 3) + ... + (1 / N), dado N =', N)

while (i != N):
    i = i + 1
    H = H + (1 / i)

print('* * * resposta: H = ', H)
```

Saída [9]:

```
* * * cálculo de H = 1 + (1 / 2) + (1 / 3) + ... + (1 / N), dado N = 5
* * * resposta: H = 2.2833333333333333
```

O comando **break** pode ser utilizado **quebrar um laço “na marra”**, passando o fluxo de execução do programa para a linha que estiver localizada imediatamente depois do fim do bloco de comandos subordinados ao laço. Por sua vez, o comando **continue** serve para **quebrar uma iteração**, mas não o laço propriamente dito. Mais claramente: sempre que o **continue** é executado, o fluxo de execução do programa é automaticamente desviado para a linha que contém o comando **while**.

Programa 10 – Quebra de laço com break e quebra de iteração com continue.

```
#P010: while com break + while com continue
print('-----')
print('1-Exemplo de while com break:\n')
n=-1;
while (n < 21):
    n=n+1;
    if n%2 != 0: break #quebra o laço se n for ímpar...
    print(n)

print('fim do while com break...\n')

print('-----')
print('2-Exemplo de while com continue:\n')
n=-1;
while (n < 21):
    n=n+1;
    if n%2 != 0: continue #quebra a iteração se n for ímpar...
    print(n)

print('fim do while com continue...\n')
```

Saída [10]:

```
-----
1-Exemplo de while com break:
```

```
0
fim do while com break...
```

```
-----
```

2-Exemplo de while com continue:

```
0
2
4
6
8
10
12
14
16
18
20
fim do while com continue...
```

Vetorização

- Os principais pacotes para ciência de dados permitem a execução de operações sobre conjuntos de dados sem a necessidade da implementação de laços. Esse processo é conhecido como **computação vetorizada** (*vectorization*) e será apresentado em detalhes a partir do Capítulo VI. Por esta razão, embora o comando **while** seja largamente utilizado na programação de sistemas convencionais, ele é bem menos empregado na ciência de dados.



Lição 9 – Instruções de Repetição (2): for – range()

Assim como o **while**, o comando **for** também serve para implementar a repetição de blocos de código. Ele existe em praticamente todas as linguagens de programação e, por ser mais prático do que o **while**, é normalmente escolhido pelos programadores em situações onde deseja-se executar um conjunto de comandos por um número fixo de vezes. Na linguagem Python o **for** possui uma característica bastante singular (e relevante!): ele pode iterar apenas sobre **sequências**. Nesta lição, mostraremos como utilizar o **for** para iterar sobre sequências criadas com a função **range()**. A partir do Capítulo III, veremos como é possível utilizá-lo para iterar sobre sequências mais complexas, como listas, dicionários, tuplas, arquivos, arrays 'NumPy' e DataFrames.

Programa 11 – Receita básica para implementar laços utilizando a estrutura for-range().

Atenção: conforme mostra o programa, o limite final definido em um **range()** não fará parte da sequência gerada (comentaremos mais sobre isso no texto após o código do programa).

```
#P011: repetição com for-range()
print('\n* * imprimindo de 0 a 9')
for i in range(10):
    print(i)

print('\n* * imprimindo de 100 a 105')
for i in range(100, 106):
    print(i)

print('\n* * 0 a 15, usando 5 como incremento')
```

```

for i in range(0, 16, 5):
    print(i)

print('\n* * ordem reversa: 5, 4, 3, 2, 1')
for i in range(5, 0, -1):
    print(i)

```

Saída [11]:

* * imprimindo de 0 a 9

```

0
1
2
3
4
5
6
7
8
9

```

* * imprimindo de 100 a 105

```

100
101
102
103
104
105

```

* * usando 5 como incremento

```

0
5
10
15

```

* * ordem reversa

```

5
4
3
2
1

```

A estrutura **for-range()** executa um laço por um número fixo de vezes. Os comandos subordinados ao **for** devem estar indentados. É necessário especificar:

- Uma **variável de controle** (ou **contador**) esta é a popular “variável do **for**”, que em nosso programa foi chamada de “i” nos quatro laços implementados.
- Os **limites inferior** e **superior** e o **valor do incremento**: esses valores determinam o número de repetições que o laço conterà e são definidos com o uso da função **range()**. A explicação sobre esta função é apresentada no tópico a seguir.

9.1 Conhecendo a Função `range()`

A função `range()` serve para gerar uma sequência de números inteiros que é geralmente utilizada para ser percorrida por um comando `for`. O Quadro 4 apresenta informações detalhadas sobre a sua sintaxe e parâmetros.

Quadro 4. Sintaxe da função `range()`

- Sintaxe: `range([início], fim, [incremento])`
- Parâmetros:
 - **início**: número inicial da sequência (opcional). Caso seja omitido, o valor 0 é assumido.
 - **fim**: a sequência será gerada até, mas sem incluir, o número especificado neste parâmetro (único parâmetro obrigatório).
 - **incremento**: diferença entre cada número na sequência (opcional). Se omitido, o valor 1 será utilizado.
- Observações:
 - Todos os parâmetros devem ser do tipo inteiro;
 - Todos os parâmetros podem ser positivos ou negativos;
 - Na função `range()` e na linguagem Python de uma maneira geral, conjuntos de valores são indexados a partir do valor 0 e não do valor 1. Em outras palavras: em Python, o primeiro elemento de uma sequência, lista, vetor, etc., sempre possuirá o índice 0. É por isso que o comando `range(n)` produz uma lista cujo primeiro elemento é 0 e o último $n-1$,
- Exemplos:
 - `range(3)` `# [0, 1, 2]`
 - `range(1, 4)` `# [1, 2, 3]`
 - `range(0, 10, 2)` `# [0, 2, 4, 6, 8]`

Para finalizar a lição, na página a seguir, o Quadro 5 apresenta um resumo com a "receita" para implementar os casos mais comuns de utilização da estrutura `for-range()`.

Quadro 5. Sintaxe da estrutura for-range() para as situações práticas mais comuns

- **for simples:** iteração de 0 a $n-1$ com incremento de 1

```
for (contador) in range(n):  
    comando1  
    ...  
    comandon
```

- **for básico:** iteração de 1 a n , com incremento de 1

```
for (contador) in range(1, n+1):  
    comando1  
    ...  
    comandon
```

- **for crescente:** vai de m até n (onde $n \geq m$), com incremento de k

```
for (contador) in range(m, n+1, k):  
    comando1  
    ...  
    comandon
```

- **for decrescente:** vai de n decrescendo até m (onde $n \geq m$), com decremento de k

```
for (contador) in range(n, m-1, -k):  
    comando1  
    ...  
    comandon
```



Lição 10 – Python ou R?

As linguagens R e Python são definitivamente as mais populares na área de ciência de dados. A escolha entre uma e outra depende basicamente do perfil da equipe de trabalho e do tipo de projeto a ser conduzido. O Quadro 6 realiza uma pequena comparação entre estas duas linguagens, tentando destacar as características mais importantes de cada uma, além de suas principais vantagens, desvantagens e formas de utilização. Você encontrará muitos textos na Internet comparando estas linguagens. No geral, há um consenso de que ambas são muito importantes e observa-se na prática que muitas empresas vêm combinando as duas em seus projetos.

Quadro 6. R versus Python

R	Python
R é uma linguagem criada por estatísticos, para estatísticos. Os manuais utilizam o vocabulário dos estatísticos e a estrutura da linguagem favorece quem possui a “mente” de estatístico. Por isso, nem sempre é de fácil assimilação para cientistas da computação.	Python é uma linguagem de cientistas da computação, para cientistas da computação. É uma linguagem multiparadigma, que incorpora todos os aspectos modernos de desenvolvimento de sistemas. Por isso costuma ser mais atraente aos olhos dos cientistas da computação.
R é uma linguagem de propósito específico: serve para análise de dados.	Python é uma linguagem de propósito geral, servindo não apenas para análise de dados, mas também para o desenvolvimento de qualquer tipo de sistema, como Web sites, jogos, sistemas corporativos, etc.
Estruturas de dados básicas para a análise de dados, como DataFrames e matrizes, fazem parte do núcleo da linguagem R.	DataFrames e matrizes não fazem parte nem do núcleo e nem da <i>standard library</i> do Python. Elas precisam ser incorporadas à linguagem através da instalação de pacotes para análise de dados como ‘NumPy’ e ‘pandas’. É ainda importante comentar que muitos pacotes populares do Python reproduzem e/ou estendem funcionalidades presentes no R, MATLAB e em outras linguagens tipicamente utilizadas para computação científica.
É mais difícil integrar os <i>scripts</i> R com os outros sistemas de uma empresa. Normalmente, torna-se preciso utilizar algum produto/componente de terceiros.	Algumas empresas preferem usar Python para ciência de dados exatamente porque a integração com os outros sistemas da empresa pode ser feita de forma direta.
R é uma linguagem interpretada e interativa.	Python também é interpretada e interativa.
Em geral, R ainda é um pouco mais utilizada para a condução de projetos de “estatística clássica” e visualização de dados.	Python costuma ser mais utilizada em projetos que envolvam inteligência artificial / <i>machine learning</i> .
A instalação e o gerenciamento de pacotes é mais simples em R: tudo é feito de forma trivial através do repositório central CRAN. Este é um fator atraente para quem não é profissional de informática.	A instalação do Python e de seus pacotes é um pouco mais difícil. Além disso, o fato de existirem muitas distribuições do Python costuma confundir as pessoas. No entanto, Python também possui um repositório central, denominado PyPi.
RStudio é, disparadamente, a IDE mais utilizada para o desenvolvimento de programas R.	Não há uma IDE que seja claramente a mais popular entre os desenvolvedores Python. Existem muitas opções como Spyder, Rodeo e IPython (muito usadas em projetos de ciência de dados) e PyDev (mais adequada para o desenvolvimento de aplicativos em geral).
R possui uma larga comunidade de usuários. Esta comunidade é extremamente ativa em fóruns como o Stack Overflow.	Python também possui uma enorme comunidade de usuários, muito atuante no Stack Overflow e em outros fóruns da Internet.

Capítulo II. Criando e Utilizando Funções

Uma função Python é nada mais do que um bloco de código que recebe um nome, o que permite com que ele possa ser chamado (isto é, executado) várias vezes em um mesmo programa. As funções foram criadas devido a uma necessidade prática comum a muitos sistemas e aplicativos: repetir uma sequência de instruções específicas em diferentes partes do programa. Para que o conceito fique claro, considere como exemplo um jogo de computador que exibe uma determinada animação toda vez que o jogador passa de fase. Como esta situação pode ocorrer diversas vezes durante uma mesma partida, o trecho de código responsável por desenhar a animação certamente seria implementado como uma função (que poderia se chamar “`animacao_passa_fase()`”, por exemplo).

Este capítulo possui dois objetivos. O primeiro é ensinar você a programar as suas próprias funções. O segundo é introduzir as funções matemáticas e estatísticas da *standard library* – um conjunto de funções prontas e muito importantes para ciência de dados que você “ganha de presente” ao instalar qualquer distribuição do Python!



Lição 11 – Criando Funções

11.1 Introdução

Uma função é um bloco de código nomeado, que pode receber um ou mais valores como entrada – denominados **parâmetros** ou **argumentos** – e que, normalmente, retorna algum resultado computado em função destes parâmetros (embora em Python, nem todas as funções precisem retornar resultados ou receber parâmetros de entrada).

A palavra reservada **def** é utilizada para a definição de funções. Mais especificamente, para criar uma função, você deve iniciar escrevendo a palavra **def**, depois indicar o nome da função, os seus parâmetros (caso existam) e, por fim, o bloco de código correspondente ao corpo da função. Dentro do corpo, o comando **return** deve ser utilizado para retornar o valor da função.

Programa 12 – Definição de uma função denominada “faixa_etaria”. Esta função recebe como entrada a idade de uma pessoa (número inteiro) e retorna como saída a sua faixa etária (string).

```
#P012: criando e utilizando uma função
def faixa_etaria(idade):
    if (idade < 18) :
        return '<18'
    elif (idade >= 18 and idade < 30) :
        return '18-29'
    elif (idade >= 30 and idade < 40) :
        return '30-39'
    else :
        return '>=40'

#chamando a função com diferentes valores para o parâmetro "idade"
a = faixa_etaria(15)
b = faixa_etaria(50)
c = faixa_etaria(35)

print(a); print(b); print(c)
```

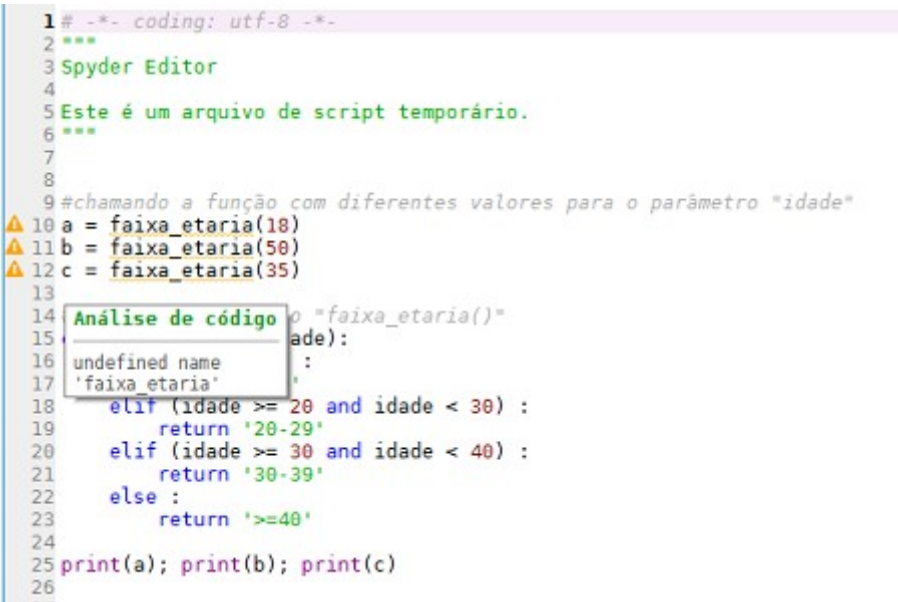
Saída [12]:

```
<18
>=40
30-39
```

Como você deve ter observado, o programa começa com a definição função “faixa_etaria()” (linhas 2-10). A primeira linha da definição da função, que contém a palavra **def**, é chamada de **cabeçalho** da função, enquanto as linhas seguintes formam o seu **corpo**. O cabeçalho deve terminar com “:” e o corpo deve estar indentado. Uma vez a função tendo sido definida, ela pode ser utilizada em qualquer parte do programa. E é exatamente isto que fizemos! No Programa 12, a função “faixa_etaria()” é chamada três vezes, preenchendo as variáveis “a”, “b” e “c”:

```
a = faixa_etaria(15)
b = faixa_etaria(50)
c = faixa_etaria(35)
```

No corpo do programa, as funções precisam ser definidas antes de serem chamadas. Por este motivo, você não pode, por exemplo, colocar o comando “faixa_etaria(35)” antes do código que define esta função. Veja que a própria IDE Spyder reclama quando você tenta fazer isso (Figura 10).



```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 Este é um arquivo de script temporário.
6 """
7
8
9 #chamando a função com diferentes valores para o parâmetro "idade"
10 a = faixa_etaria(18)
11 b = faixa_etaria(50)
12 c = faixa_etaria(35)
13
14 def faixa_etaria(idade):
15     if (idade < 20):
16         return '<18'
17     elif (idade >= 20 and idade < 30):
18         return '20-29'
19     elif (idade >= 30 and idade < 40):
20         return '30-39'
21     else:
22         return '>=40'
23
24
25 print(a); print(b); print(c)
26
```

Análise de código

undefined name 'faixa_etaria'

Figura 10. Função definida em local errado

11.2 Módulos

Existem duas formas de utilizar uma função. A primeira é criá-la e executá-la em um mesmo programa como acabamos de fazer. A segunda consiste em defini-la em um arquivo separado do programa principal e depois importá-la com o uso do comando **import**. No mundo Python, um arquivo que contém apenas funções recebe o nome de **módulo**.

Programa 13 – Importação de Módulo. Se você salvar um arquivo chamado “minhas_funcoes.py”, contendo a definição da função “faixa_etaria()”, poderá importá-lo e utilizá-lo como um módulo Python empregando a forma mostrada no programa abaixo.

```
#P013: importando e utilizando uma função de um módulo
import minhas_funcoes
```

```
a = minhas_funcoes.faixa_etaria(15)
b = minhas_funcoes.faixa_etaria(50)
c = minhas_funcoes.faixa_etaria(35)
```

```
print(a); print(b); print(c)
```

Saída [13]:

```
<18
>=40
30-39
```

Observações:

- Como mostrado, a forma de utilizar uma função de um módulo é um pouco diferente: você deve escrever o nome do módulo, depois um ponto (".") e o nome da função (ex.: `minhas_funcoes.faixa_etaria(18)`).
- O nome do arquivo referente ao módulo deve possuir a extensão ".py" (ex.: "minhas_funcoes.py"). No entanto, ao importar o módulo o ".py" não deve ser especificado.
- Um módulo pode conter uma ou mais funções.
- O arquivo deve ser salvo na mesma pasta do(s) programa(s) que farão uso do mesmo ou em alguma que seja visível por qualquer outra pasta (neste caso, no ambiente Windows, trata-se de uma pasta que tenha sido especificada na variável de ambiente PATH).

Pacotes

- A linguagem Python permite com que você agrupe coleções de módulos que contenham funcionalidades similares ou relacionadas em uma estrutura denominada **pacote**. Os pacotes também podem ser importados por outros programas através do comando **import**.
- Neste livro, não abordaremos a criação de pacotes. Entretanto, em capítulos subsequentes, mostraremos como trabalhar com diversos pacotes prontos disponibilizados pelo WinPython ou obtidos através do PyPI.

11.3 Parâmetros Opcionais e o Valor None

Funções podem ser definidas com **parâmetros opcionais**, como mostra o próximo exemplo.

Programa 14 – Função com parâmetro opcional. Neste programa, a função "soma_numeros()" possui três parâmetros ("x", "y" e "z"), mas o terceiro recebe o valor **None** como *default*. Com isto, você poderá chamar a função passando dois ou três números como parâmetro. No primeiro caso, ela retornará $x + y$ e no segundo $x + y + z$.

```
#P014: função com parâmetro opcional
def soma_numeros(x,y,z=None):
    if (z==None):
        return x+y
    else:
        return x+y+z

print(soma_numeros(1, 2))
print(soma_numeros(1, 2, 3))
```

Saída [14]:

```
3
6
```

Mas o que é None? Trata-se de um valor especial, que possui o seu próprio tipo ("NoneType") e que pode ser entendido como "ausência de qualquer coisa". Voltando aos parâmetros opcionais, é importante observar que estes devem sempre ser colocados como os últimos parâmetros na declaração da função. Além de None, você pode atribuir qualquer outro valor *default* para os parâmetros opcionais. Veja o exemplo a seguir.

Programa 15 – Função com parâmetro que possui valor *default*. Definição da função “f_calcula()”, cujo terceiro parâmetro (“operacao”) possui o valor “+” como *default*. Se o usuário chamar a função sem passar o terceiro parâmetro, o valor “+” será automaticamente adotado.

#P015: outra função com parâmetro que possui valor default

```
def f_calcula(x,y,operacao='+'):  
    if (operacao=='+'):   
        return x+y  
    elif (operacao=='-'):   
        return x-y  
    elif (operacao=='*'):   
        return x*y  
    elif (operacao=='/'):   
        return x/y  
    else:  
        return 'operação inválida!'  
  
print(f_calcula(1, 2))           #retorna 1+2 = 3  
print(f_calcula(1, 2, '+')) #retorna 1+2 = 3  
print(f_calcula(1, 2, '-')) #retorna 1-2 = -1  
print(f_calcula(1, 2, '*')) #retorna 1*2 = 2  
print(f_calcula(1, 2, '/')) #retorna 1/2 = 0.5  
print(f_calcula(1, 2, '.')) #retorna 'operação inválida'
```

Saída [15]:

```
3  
3  
-1  
2  
0.5  
operação inválida!
```

11.4 Procedimentos

Nos exemplos anteriores, todas as funções retornaram algum valor através do comando **return**. No entanto, também é possível utilizar o comando **def** para criar funções que apenas executam algum tipo de ação, mas não retornam nenhum valor para o usuário. Essas funções não possuem o comando **return**.

Programa 16 – Definição de função que não retorna valor. O exemplo a seguir ilustra um programa Python que possui a declaração da função “imprime_cabecalho()”. Considere que ela seja utilizada para imprimir a string passada em “nome” como o cabeçalho para os relatórios de uma empresa hipotética. Ela não retorna qualquer valor ao ser chamada! Em vez disso, simplesmente executa uma ação: imprimir o cabeçalho.

#P016: função que não retorna valor

```
def imprime_cabecalho(nome):  
    print("\n-----")  
    print("* * Relatório - " + nome)  
    print("-----\n")  
  
imprime_cabecalho('Produtos Mais Vendidos')  
imprime_cabecalho('Produtos Esgotados')
```

Saída [16]:

```
-----  
* * Relatório - Produtos Mais Vendidos  
-----
```

```
-----  
* * Relatório - Produtos Esgotados  
-----
```

Em outras linguagens de programação, um módulo reutilizável que não retorna valor é chamado de procedimento (*procedure*) ou *void*. Mas em Python tudo é considerado função, independente de retornar valor ou não. Podemos nos certificar fazendo a verificação do tipo de “`imprime_cabecalho()`”:

```
print(type(imprime_cabecalho))
```

Saída [16]:

```
<class 'function'>
```

Também é importante deixar claro que é possível criar funções sem parâmetro. Por exemplo, se todos os cabeçalhos dos relatórios da empresa possuísem uma mensagem fixa, a função “`imprime_cabecalho()`” não precisaria ter nenhum parâmetro, como mostra o código abaixo:

```
def imprime_cabecalho():  
    print("\n-----")  
    print("* * * * * Relatório * * * * *")  
    print("-----\n")
```

11.5 Valores de Parâmetros

Em Python, quando uma variável de um tipo primitivo (inteiro, float, string ou boolean) é passada como argumento de uma função e tem o seu conteúdo alterado dentro do bloco de código, a alteração em questão **não é refletida** para o programa principal (para quem chamou a função). Para facilitar o entendimento desse conceito, apresentaremos um programa exemplo que será comentado detalhadamente.

Programa 17 – Função recebendo uma variável de tipo primitivo como argumento.

```
#P017: passagem de parâmetro - variável de tipo primitivo  
def soma_um(numero):  
    numero=numero+1  
    print("somei um dentro da função: ", numero)  
  
k=100  
print('valor original de k:', k)  
soma_um(k)  
print('Terminou a função e k, na verdade, não mudou:', k)
```

Saída [17]:

```
valor original de k: 100  
somei um dentro da função: 101  
Terminou a função e k, na verdade, não mudou: 100
```

Explicação:

- Inicialmente, temos a definição de uma função bem simples, chamada "soma_um(numero)", uma *função-estilo-procedimento*, ou seja, que não retorna valor. Esta função recebe como entrada um número, soma um a este número e, depois da soma, imprime seu valor.
- Então começa o programa principal, com as instruções `k=100` e `print('valor original de k:', k)`. Nenhuma novidade até aí! Vamos apenas aproveitar para introduzir um novo conceito: "k" é chamada de **variável global**, pois é uma variável declarada no corpo do programa principal. Uma variável declarada dentro de uma função é, por sua vez, chamada de **variável local**.
- Em seguida vem o comando: `soma_um(k)`, em que a variável "k" é passada como argumento para "soma_um(numero)". Com isto, o valor de "numero" passa a ser o mesmo de "k", ou seja, 100. Dentro da função, "numero" é incrementado em uma unidade (passa para 101 – o conteúdo de "numero" é impresso para não deixar nenhuma dúvida!).
- Entretanto, quando a função "morre" (ao término de sua execução), veja que esta alteração não foi refletida para "k" (como mostra o último `print()` do programa principal). Mais claramente: embora "numero" tenha mudado de 100 para 101 dentro da função "soma_um(numero)", o valor de "k" continuou igual a 100 fora desta função.

Mas por que aconteceu assim? A resposta é bem simples. Isto ocorre porque na linguagem Python todos os parâmetros de funções que estejam associados a tipos primitivos são sempre tratados como **parâmetros de valor**. E sempre que uma variável é passada como parâmetro de valor para uma função, gera-se automaticamente uma **cópia temporária** da mesma em memória. Durante a execução da função, somente essa cópia temporária é usada. Deste modo, quando o valor do parâmetro é alterado, isso só tem efeito sobre o armazenamento temporário (o efeito é local à função). A variável fora do procedimento (variável global, nosso "k" do exemplo) jamais será tocada.

11.6 *args

Este recurso do Python nos permite passar um número arbitrário de parâmetros para uma função. Veja o exemplo a seguir:

Programa 18 – função com número arbitrário de parâmetros

```
#P018: *args
def pessoa(nome, *args):
    print("- nome (primeiro parâmetro): ", nome)
    print("- características (outros parâmetros): ")
    for arg in args:
        print("\t", arg)

pessoa('Jane', 'escritora', 'sagitariana', 'romântica')
print("\n")
pessoa('John', 'músico')
```

Saída [18]:

- nome (primeiro parâmetro): Jane
- características (outros parâmetros):

escritora
sagitariana
romântica

- nome (primeiro parâmetro): John
- características (outros parâmetros): músico

Na função “`pessoa(nome, *args)`”, o parâmetro “nome” é como qualquer outra que já mostramos. No entanto, “`*args`” pode receber múltiplos parâmetros. Veja que na primeira chamada da função, “`*args`” recebeu 3 parâmetros (“escritora”, “sagitariana” e “romântica”), enquanto na segunda apenas 1 (“músico”).

É importante observar que para a definição de múltiplos parâmetros o que vale mesmo é o `*`. Isto quer dizer que poderíamos ter assinado a função como “`pessoa(nome, características*)`”. No entanto, a maioria dos programadores Python adota “`args*`” como convenção.



Lição 12 – Funções Pré-Definidas

As funções pré-definidas (*built-in functions*) são aquelas que fazem parte da própria linguagem Python, podendo ser utilizadas sem que seja preciso importar qualquer módulo. No capítulo anterior trabalhamos com três delas: `type()`, `print()` e `input()`. No Quadro 7, são apresentadas mais nove funções pré-definidas. Na notação utilizada, considere que `x` é um parâmetro do tipo inteiro ou float e que `s` é um parâmetro do tipo string:

Quadro 7. Funções pré-definidas

Funções Numéricas

- **`abs(x)`** : retorna o valor absoluto de `x`;
- **`pow(x, y)`** : retorna `x` elevado a `y`.
- **`round(x, d)`** : retorna `x` arredondado para `d` casas decimais (neste caso, `x` deve ser um float).

Funções de String

- **`len(s)`** : retorna o comprimento (número de caracteres) de `s`;
- **`max(s)`** : retorna o maior caractere de `s`, considerando a ordem lexicográfica;
- **`min(s)`** : retorna o menor caractere de `s`, considerando a ordem lexicográfica.

Funções de Conversão de Tipo

- **`float(s)`** : converte a string `s` para um float. A variável `s` deve conter um número válido, inteiro ou float, caso contrário a função retornará um erro;
- **`int(s)`** : converte a string `s` para um inteiro. A variável `s` deve conter um número inteiro válido, caso contrário a função retornará um erro;
- **`str(n)`** : converte o número `n` para uma string. Seu uso é necessário quando você quer concatenar uma string com um número.

Programa 19 – Exemplifica o uso das funções pré-definidas.

```
#P019: funções pré-definidas

#funções numéricas
n1=100
n2=3.141592653
n3=9.99

print(abs(1000), abs(-500), abs(2 * -2.5), abs(0)) #1000 500 5.0 0
print(pow(n1,2)) #10000
print(round(n2,2)) #3.14
print(round(n2), round(n3)) #3 10

#funções de conversão
s1='5'
s2='9.99'

print(int(s1)) #converteu '5' -> 5
print(float(s2)) #converteu '9.99' -> 9.99
print('O valor de PI com 10 digitos é: ' + str(n2))
print('O valor de PI com 2 digitos é: ' + str(round(n2,2)))

#funções de string
s1='python'
s2='inconstitucional'

print(len(s1)) #6
print(len(s2)) #16
print(max(s1)) #'y'
print(min(s1)) #'h'
```

Saída [19]:
1000 500 5.0 0
10000
3.14
3 10
5
9.99
O valor de PI com 10 digitos é: 3.141592653
O valor de PI com 2 digitos é: 3.14
6
16
y
h



Lição 13 – Funções *lambda*

Uma função *lambda* é uma pequena função anônima que deve possuir apenas **uma expressão** definida em uma **única linha** de comando. Trata-se de uma **notação abreviada** que pode ser empregada para definir funções simples (uma expressão, uma linha) com o uso da palavra reservada **lambda**. A seguinte sintaxe é utilizada: **lambda** *parâmetros: expressão*. Por exemplo:

```
lambda x,y: x + y
```

Esta é uma função lambda com dois parâmetros (“x” e “y”) que retorna a soma de ambos. Veja que **não** há a necessidade de utilizar o comando **return**. Além disso, observe que a função possui uma única linha contendo apenas uma expressão. Com relação ao número de parâmetros, não existe limite, podemos definir quantos a gente quiser.

Você pode atribuir uma função lambda a uma variável e então utilizá-la sempre que for necessário dentro de um programa. Veja o exemplo a seguir:

Programa 20 – Função lambda.

```
#P020: função lambda
soma = lambda x,y: x + y
print(soma(1,2)) #3
print(soma(5,10)) #15
```

Saída [19]:

```
3
15
```

Por enquanto vamos parar por aqui, mas ao longo do livro mostraremos outras aplicações interessantes para as funções lambda. Caso você esteja curioso, um artigo bem detalhado a respeito do tema pode ser encontrado no endereço <https://realpython.com/python-lambda>.



Lição 14 – Módulo ‘math’

O módulo ‘math’ é um dos muitos que fazem parte da *standard library*. Portanto, ele é automaticamente instalado por qualquer distribuição do Python (CPython, Anaconda, WinPython, etc.). Este módulo fornece uma série de funções e constantes matemáticas úteis, tais como funções de arredondamento, trigonométricas, logarítmicas, etc. Antes de utilizá-lo, você precisa realizar a sua importação da seguinte maneira:

```
import math
```

Nos Quadros 8 e 9 são relacionadas, respectivamente, algumas das principais constantes e funções disponibilizadas pelo módulo ‘math’¹¹. Em seguida, apresenta-se um programa que exemplifica o uso das mesmas.

Quadro 8. Constantes do módulo ‘math’

- `math.pi`: constante matemática 3.14159...;
- `math.e`: constante matemática 2.718281... (número de Euler);
- `math.tau`: constante matemática 6.283185... (equivalente a 2π);
- `math.inf`: valor float que representa $+\infty$. Para $-\infty$ basta usar `-math.inf`;

11 Documentação completa do módulo ‘math’ em: <https://docs.python.org/3/library/math.html>

- `math.nan`: trata-se do famoso NaN, valor float que representa “*not a number*”. Este valor é gerado pelo Python sempre que o resultado de um cálculo não puder ser expresso por um número. Qualquer cálculo envolvendo NaN sempre resultará em NaN (ex.: `1 + NaN = NaN`).

Quadro 9. Funções do módulo 'math'

Funções de Arredondamento

- `math.ceil(x)`: arredondamento “pra cima”, ou seja, retorna o menor inteiro com valor igual ou superior a x ;
- `math.floor(x)`: arredondamento “pra baixo”, ou seja, retorna o maior inteiro com valor igual ou inferior a x .
- `math.trunc(x)`: truncamento, o que significa limitar o número de dígitos de x .

Funções Logarítmicas / Exponenciais

- `math.exp(x)`: retorna e elevado a x ;
- `math.log2(x)`: retorna o logaritmo de x na base 2;
- `math.log10(x)`: retorna o logaritmo de x na base 10;
- `math.log(x, b)`: retorna o logaritmo de x na base b (de acordo com a documentação do Python, deve ser utilizada apenas quando b é diferente de 2 e 10).
- `math.pow(x, y)`: retorna x elevado a y ;
- `math.sqrt(x)`: retorna a raiz quadrada de x ;

Funções Trigonômicas (em todas elas, x é um ângulo que deve ser fornecido em radianos)

- `math.acos(x)`: retorna o arco cosseno de x ;
- `math.asin(x)`: retorna o arco seno de x ;
- `math.atan(x)`: retorna o arco tangente de x ;
- `math.cos(x)`: retorna o cosseno de x ;
- `math.sin(x)`: retorna o seno de x ;
- `math.tan(x)`: retorna a tangente de x ;
- `math.degrees(x)`: converte o ângulo x de radianos para graus;
- `math.radians(g)`: converte o ângulo g de graus para radianos.

Funções Hiperbólicas

- Também existem as funções hiperbólicas análogas, cujos os nomes sempre terminam com letra “h”. Ex.: `math.tanh(x)` retorna a tangente hiperbólica de x .

Funções para Teste de Valores

- `math.isnan(x)`: retorna True caso x seja NaN; caso contrário False é retornado;
- `math.isfinite(x)`: retorna True caso x não seja infinito e nem NaN; caso contrário, False é retornado;
- `math.isinf(x)`: retorna True caso x seja infinito (positivo ou negativo); caso contrário, False é retornado,

Programa 21 – Exemplifica o uso de constantes e funções do módulo 'math'.

```
#P021: módulo 'math'
import math

#constante PI
print('PI=',math.pi)          #3.141592653589793

#funções de arredondamento
x1 = 5.9
print('\n')
print(x1)
print('ceil',math.ceil(x1))
print('floor',math.floor(x1))
print('trunc',math.trunc(x1))

#logaritmo
print('\n')
x2 = 1024
print('log de',x2,'na base 2: ', math.log2(x2))

#imprime tabela com seno, cosseno e tangente de 30, 45 e 60
#note que é preciso converter os ângulos para radianos
print('\n')
for angulo_graus in range(30,61,15):
    angulo_radianos = math.radians(angulo_graus)
    print('\n* * * Angulo=',angulo_graus, ' graus')
    print('SENO=',round(math.sin(angulo_radianos),2))
    print('COSSENO=',round(math.cos(angulo_radianos),2))
    print('TANGENTE=',round(math.tan(angulo_radianos),2))
```

Saída [21]:

PI= 3.141592653589793

5.9
ceil 6
floor 5
trunc 5

log de 1024 na base 2: 10.0

* * * Angulo= 30 graus
SENO= 0.5
COSSENO= 0.87
TANGENTE= 0.58

* * * Angulo= 45 graus

```
SENO= 0.71
COSSENO= 0.71
TANGENTE= 1.0
```

```
* * * Angulo= 60 graus
SENO= 0.87
COSSENO= 0.5
TANGENTE= 1.73
```



Lição 15 – Módulo 'statistics'

Este módulo fornece funções para cálculos estatísticos básicos¹² sobre conjuntos de dados. Os dados a serem processados devem ser especificados como listas¹³ de valores. São oferecidas funções para o cálculo das medidas de tendência central e de variabilidade, conforme apresentado no Quadro 10 (considere que o parâmetro “lst” é um conjunto de dados representado em uma lista).

Quadro 10. Funções do módulo 'statistics'

- `statistics.mean(lst)`: média dos valores de *lst*;
- `statistics.median(lst)`: mediana de *lst* (uma coisa legal é que os valores não precisam estar ordenados);
- `statistics.harmonic_mean(lst)`: média harmônica;
- `statistics.mode(lst)`: moda de uma lista de valores discretos ou categóricos;
- `statistics.median_low(lst)`: quando *lst* possui um número par de valores, retorna o menor dos dois valores que seriam usados para computar a mediana. Exemplo: para [1,2,3,4], retorna 2;
- `statistics.median_high(lst)`: quando *lst* possui um número par de valores, retorna o maior dos dois valores que seriam usados para computar a mediana. Exemplo: para [1,2,3,4], retorna 3;
- `statistics.median_grouped(lst)`: retorna a mediana de dados contínuos agrupados, calculada como o 2º quartil, usando interpolação;
- `statistics.stdev(lst)`: desvio padrão de amostra;
- `statistics.pstdev(lst)`: desvio padrão de população;
- `statistics.variance(lst)`: variância da amostra;
- `statistics.pvariance(lst)`: variância da população;

Programa 22 – Exemplifica o uso das funções do módulo 'statistics'. IMDb (<https://www.imdb.com/>) é um conhecido Web site sobre cinema que armazena informações completas sobre centenas de milhares de filmes. Entre estes filmes, encontram-se os que foram

12 Documentação completa do módulo 'statistics' em: <https://docs.python.org/3/library/statistics.html>

13 Aprenderemos sobre listas no próximo capítulo.

dirigidos por Milos Forman¹⁴, cineasta que ganhou o Oscar de melhor diretor em duas ocasiões. O IMDb permite com que seus usuários atribuam notas para os filmes, que podem variar de 0 a 10. Quando um usuário busca informações sobre um filme, o IMDb também exibe a nota média de avaliação dos usuários. No caso de Milos Forman temos as seguintes avaliações médias para os filmes por ele dirigidos depois de 1970:

Filme	Nota do Público (média)
Procura Insaciável (1971)	7.4
Um Estranho no Ninho (1975)	8.7
Hair (1979)	7.6
Na Época do Ragtime (1981)	7.3
Amadeus (1984)	8.3
Valmont - Uma História de Seduções (1989)	7.0
O Povo Contra Larry Flint (1996)	7.3
O Mundo de Andy (1999)	7.4
Sombras de Goya (2006)	6.9
Dobre placená procházka (2009)	6.7

O programa listado abaixo cria duas listas de valores, denominadas “nomes_filmes” e “avaliacao_filmes”, que recebem os nomes e as avaliações dos filmes, respectivamente. Mostramos então como usar o módulo 'statistics' para computar a média, mediana, variância e desvio padrão das avaliações. Observe que um apelido foi atribuído para o módulo 'statistics' no momento da importação (**import statistics as s**) para que fosse possível utilizar suas funções escrevendo simplesmente a letra 's' em vez de 'statistics' (apelidar não é obrigatório, fizemos isso apenas porque 'statistics' é uma palavra muito grande!).

```
#P022: módulo 'statistics'
import statistics as s

nomes_filmes = ['Procura Insaciável (1971)',
                'Um Estranho no Ninho (1975)',
                'Hair (1979)',
                'Na Época do Ragtime (1981)',
                'Amadeus (1984)',
                'Valmont - Uma História de Seduções (1989)',
                'O Povo Contra Larry Flint (1996)',
                'O Mundo de Andy (1999)',
                'Sombras de Goya (2006)',
                'Dobre placená procházka (2009)']
avaliacao_filmes = [7.4, 8.7, 7.6, 7.3, 8.3, 7.0, 7.3, 7.4, 6.9, 6.7]

print('média = ',s.mean(avaliacao_filmes))
print('mediana = ',s.median(avaliacao_filmes))
print('variância = ',s.variance(avaliacao_filmes))
print('desvio padrão = ',s.stdev(avaliacao_filmes))
```

Saída [22]:

```
média = 7.46
mediana = 7.35
variância = 0.3804444444444443
desvio padrão = 0.61680178699842
```

14 <https://www.imdb.com/name/nm0001232/>



Lição 16 – Módulo 'random'

Este módulo oferece uma série de funções para a geração de números aleatórios. Alguns exemplos são apresentados no Quadro 11:

Quadro 11. Funções do módulo 'random'

- `random.random()`: retorna um número real (float) na faixa [0.0, 1.0). Esta notação indica que 0.0 pode ser gerado, mas 1.0 não;
- `random.randint(inicio, fim)`: retorna um número inteiro na faixa [inicio, fim);
- `random.choice(sequência)`: sorteia um elemento de uma sequência;
- `random.seed(semente)`: utilizada para definir uma semente em experimentos reproduzíveis. Quando a semente não é especificada, a mesma é atribuída conforme o *timestamp* do sistema (data e hora com precisão de nanossegundos);
- `random.getstate()`: captura o estado corrente do gerador de números aleatórios (mais informações no Programa 21).
- `random.setstate(e)`: utilizado para retornar o gerador para o estado *e*, previamente capturado pela função `getstate()`.

Programa 23 – Exemplifica o uso das funções do módulo 'random'. Preste especial atenção na receita que deve ser utilizada para trabalhar com sementes.

```
#P023: módulo 'random'
import random

#(1)-função randint()
for i in range(6):
    n = random.randint(1,61) #próximos números da mega-sena???
    print(n)

#(2)-função choice()
k = random.choice([1,2,3,4,5])
print("\n", k)

#(3)-trabalhando com sementes
#quando a semente não é especificada, o Python usa o timestamp do sistema
#e os valores de n1 e n2 mudarão sempre que o programa for executado
n1=random.random()
n2=random.random()
print("\n", n1, n2)

#Mas se você especifica a semente, a série de números será sempre a mesma:
estado = random.getstate()
random.seed(2020)

x1=random.random()    #0.6196...
x2=random.random()    #0.1745...
print(x1, x2)
```

```
random.setstate(estado)
```

```
#IMPORTANTE: para poder "desabilitar" a semente (voltar ao valor default, ou seja  
#o timestamp do sistema), é preciso utilizar a receita mostrada acima:  
#  
# 1 -> capturar o estado default do gerador (que é o timestamp do sistema),  
# usando getstate()  
# 2 -> mudar a semente com seed()  
# 3 -> retornar ao estado default do gerador usando setstate
```

Saída [23]:

```
16  
13  
41  
15  
58  
3
```

```
1
```

```
0.1475140766164711 0.767479049912807  
0.6196692706606616 0.17452386521097274
```

Vale a pena acessar a documentação¹⁵ completa do módulo ‘random’ para conhecer o restante de suas funções.

Módulos Matemáticos

- É importante saber que há dois outros módulos matemáticos na standard library: ‘fractions’ (implementação de operações numéricas para números racionais) e ‘decimal’ (aritmética de números reais com representação em ponto fixo ou flutuante). Isto sem falar na biblioteca ‘NumPy’, tema do Capítulo VI, que é responsável por oferecer um rico conjunto de ferramentas voltadas para a manipulação eficiente de vetores e matrizes. .



Lição 17 – Função print()

A função `print()` serve para imprimir um ou mais objetos (mensagens, números, listas, etc.) no dispositivo de saída padrão, que normalmente é o monitor de vídeo do computador. Esta função foi utilizada em todos os exemplos de código apresentados até agora, mas apenas de uma forma básica. A seguir, apresenta-se um programa que mostra outras possibilidades de utilização.

Programa 24 – Parâmetros e operadores que podem ser utilizados para enriquecer as funcionalidades de `print()`.

¹⁵ Na verdade, os números gerados por ‘random’ são chamados de números pseudoaleatórios, já que são gerados por computação determinística. Documentação completa do módulo ‘random’ em: <https://docs.python.org/3/library/random.html>


```

#P024: print() revisitado
a=1; b=2; c=3

#imprime os valores de "a", "b" e "c" separados por espaço
print(a,b,c)          #1 2 3

#parâmetro "sep": troca espaço pelo separador especificado
print(a,b,c,sep=';')   #1;2;3

#parâmetro "end": por padrão, a função termina uma linha com \n (quebra de linha)
#Mas você pode terminar com qualquer caractere usando o parâmetro "sep"
print("Meu Primeiro",end=' ')
print("Livro",end=' ')
print("de Python")

#a função format() facilita a impressão e formatação de dados de variáveis
#consulte: https://pyformat.info/
nome='PI'
v=3.14159;
print("{} com duas casas decimais={:.2f}".format(nome,v)) #PI ... 3.14

#o operador especial % permite a formatação no estilo "Linguagem C"
#consulte: https://www.learnpython.org/en/String_Formatting.
print("PI convertido para string=%s" % v)                  #'3.14159'

#o sinal de escape (\) também permite a impressão de aspas
print("Imprimindo aspas \"")          #imprimindo aspas "
print('Imprimindo aspas \'')          #imprimindo aspas '

#adicionando quebras de linha com \n
print("linha1\nlinha2\nlinha3")

#separando valores por tabulação com \t
print("coluna1\tcoluna2\tcoluna3")

```

Saída [24]:

```

1 2 3
1;2;3
Meu Primeiro Livro de Python
PI com duas casas decimais=3.14
PI convertido para string=3.14159
Imprimindo aspas "
Imprimindo aspas '
linha1
linha2
linha3
coluna1 coluna2 coluna3

```



Lição 18 – Função help()

A função **help()** exibe um texto de ajuda sobre um determinado comando, função ou método da linguagem Python.

```
help(len)           #mostra o help da função len()
```

Help on built-in function len in module builtins:

```
len(obj, /)
```

Return the number of items in a container.

```
import statistics
```

```
help(statistics.median_low) #mostra o help da função 'median_low' do módulo  
                             # 'statistics' (o módulo precisa ser importado)
```

Help on function median_low in module statistics:

```
median_low(data)
```

Return the low median of numeric data.

When the number of data points is odd, the middle value is returned.

When it is even, the smaller of the two middle values is returned.

```
median_low([1, 3, 5])
```

3

```
median_low([1, 3, 5, 7])
```

3

Capítulo III. Estruturas de Dados Nativas

As estruturas de dados (EDs) são utilizadas para organizar e armazenar conjuntos de dados relacionados, com o intuito de permitir com que estes sejam processados de forma eficiente por diferentes algoritmos. No papel de cientista de dados, você frequentemente precisará trabalhar com EDs para conseguir estudar ou transformar bases de dados.

As EDs podem ser divididas em duas categorias: primitivas e não-primitivas (estas últimas também chamadas de complexas ou agregadas). As estruturas primitivas nós já conhecemos muito bem: são simplesmente as variáveis comuns que armazenam valores atômicos (isto é, valores simples e indivisíveis) do tipo `int`, `float`, `string` ou `boolean`. Exemplos: uma variável `string` que armazena o nome de uma pessoa e uma variável do tipo `float`, que armazena o salário da mesma.

As estruturas agregadas, por outro lado, são bem mais avançadas. Elas são capazes de armazenar coleções de valores relacionados, organizando-os sempre de alguma maneira “esperta” em diferentes células. Um exemplo de estrutura deste tipo seria uma tabela em memória contendo os nomes e salários de todos os funcionários de uma empresa, com os dados dispostos em ordem crescente de salário. Nas linguagens de programação modernas como Python, as EDs agregadas costumam vir acompanhadas de diversos métodos úteis para facilitar a sua operação (“método” é somente um nome mais estiloso para função!).

Neste capítulo, você aprenderá a trabalhar com as quatro estruturas agregadas nativas da linguagem Python: listas, tuplas, conjuntos e dicionários. Depois de concluir o capítulo, você terá um sólido entendimento sobre as principais características de cada uma delas, além de seus principais tipos de operações e casos práticos de uso. Esse conhecimento é muito importante para dominar os pacotes específicos para ciência de dados do Python, como ‘NumPy’ e ‘pandas’, que oferecem outras EDs agregadas ainda mais sofisticadas.



Lição 19 – Listas

19.1 Introdução

Uma lista é uma **sequência ordenada de elementos**, cada qual associado a um número responsável por indicar a sua posição. Este número é denominado **índice**. O primeiro índice de uma lista é sempre 0, o segundo 1, e assim por diante. As listas do Python são parecidas com os vetores de outras linguagens de programação, como Java ou C, sendo, porém, um pouco mais flexíveis. Para criar uma lista, basta especificar uma sequência de valores entre colchetes, onde os valores devem estar separados por vírgula. Veja alguns exemplos:

```
escritores = ['Jorge Amado', 'José Saramago', 'Aldous Huxley']

sequencia_fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

lista_vazia = []
```

Os elementos de uma lista não precisam ser do mesmo tipo. Por exemplo, a lista a seguir contém valores do tipo string, float e até mesmo uma sublista:

```
lst_mista = ['Ana', 9.5, [10, 20, 30]]
```

Também é possível usar a função **list()** para criar uma lista a partir de uma string (e de outros tipos de sequência):

```
lst = list("Recife")    #gera a lista ['R','e','c','i','f','e']
```

19.2 Operações sobre Listas

Programa 25 – Operações de Indexação, Fatiamento, Iteração, Busca e Modificação em Listas.

Este programa realiza a definição de uma lista denominada “lst_bichos” cujos elementos são alguns bichinhos que vivem no Rio de Janeiro e não são tão assustadores como a cobra píton! O programa apresenta diversas formas distintas para acessar os elementos e para gerar sublistas que representam “recortes” da lista. A primeira operação é conhecida como **indexação** (*indexing*) e a segunda como **fatiamento** (*slicing*). Também apresentamos a forma básica para percorrer uma lista em sequência (operação de **iteração**) e para buscar e modificar os seus elementos.

```
#P025: processamento básico de listas
lst_bichos = ['bem-te-vi', 'capivara', 'saracura', 'teiú']

tam_lista = len(lst_bichos)    #nossa conhecida função len() pode ser utilizada p/
                                #obtermos o tamanho (número de elementos) da lista

#(1)-indexando os elementos
primeiro = lst_bichos[0]        #retorna 'bem-te-vi'
ultimo = lst_bichos[3]          #retorna 'teiú'
ultimo_tambem = lst_bichos[-1]  #também retorna 'teiú'

print('lst_bichos: ', lst_bichos)
```

```

print('tipo de dado: ', type(lst_bichos))
print("total de elementos: ", tam_lista)
print("primeiro elemento: " + primeiro)
print("último elemento: " + ultimo)
print("último elemento de novo: " + ultimo_tambem)
print('-----')

#(2)-iteração: percorrendo os elementos com um laço for-in
k=1;
for b in lst_bichos:
    print('elemento ', k, ' = ', b)
    k = k+1
print('-----')

#(3)-fatiamento: obtendo sublistas (slices)
print(lst_bichos[0:2])          #['bem-te-vi','capivara']
print(lst_bichos[2:4])          #['saracura','teiú']
print(lst_bichos[:2])           #['bem-te-vi','capivara']
print(lst_bichos[2:])           #['saracura','teiú']
print('-----')

#(4)-operador in (busca): elemento pertence à lista?
tem_capivara = 'capivara' in lst_bichos
tem_piton = 'píton' in lst_bichos

print("'capivara' está na lista? -> ", tem_capivara)
print("'píton' está na lista? -> ", tem_piton)
print('-----')

#(5)-modificando o conteúdo
lst_bichos[2] = 'sabiá'          #altera o elemento de índice 2
print("nova lista -> ", lst_bichos)
lst_bichos[:2] = ['bicudo','preá'] #altera os elementos 0 e 1
print("novíssima lista -> ", lst_bichos)

```

Saída [25]:

```

lst_bichos: ['bem-te-vi', 'capivara', 'saracura', 'teiú']
tipo de dado: <class 'list'>
total de elementos: 4
primeiro elemento: bem-te-vi
último elemento: teiú
último elemento de novo: teiú
-----
elemento 1 = bem-te-vi
elemento 2 = capivara
elemento 3 = saracura
elemento 4 = teiú
-----
['bem-te-vi', 'capivara']
['saracura', 'teiú']
['bem-te-vi', 'capivara']
['saracura', 'teiú']
-----
'capivara' está na lista? -> True
'píton' está na lista? -> False
-----
nova lista -> ['bem-te-vi', 'capivara', 'sabiá', 'teiú']

```

novíssima lista -> ['bicudo', 'preá', 'sabiá', 'teiú']

A seguir apresenta-se uma breve explicação sobre o programa, que está dividido em cinco partes:

- A primeira mostra que colchetes "[]" são utilizados para acessar elementos de uma lista. Dentro dos mesmos, deve-se especificar um número inteiro que corresponde ao índice do elemento que você deseja acessar. Se você especificar um número negativo, o Python faz a indexação de trás pra frente, isto é -1 pega o último elemento da lista, -2 o penúltimo, etc.
- Na segunda parte, mostramos como utilizar a estrutura **for...in** para percorrer uma lista. O bloco de código subordinado ao laço será executado uma vez para cada elemento da lista. Observe que a cada iteração, a variável "b" (variável de iteração) recebe automaticamente um dos elementos da lista.
- A terceira parte é a mais interessante, merecendo assim comentários detalhados. Ela demonstra a notação utilizada pelo Python para a operação de fatiamento. Neste tipo de operação, você pode obter uma sublista a partir de uma lista qualquer utilizando a sintaxe apresentada no Quadro 12.

Quadro 12. Sintaxe da operação de fatiamento de listas - situações práticas mais comuns

- **Operação de fatiamento de uma lista "lst":**
- **lst[i:j]**: do elemento de índice *i* ao de índice *j-1*.
- **lst[i:]**: do elemento de índice *i* até o último da lista.
- **lst[:j]**: do primeiro elemento da lista (índice 0) ao elemento de índice *j-1*.
- **lst[i:j:k]**: do elemento de índice *i*, até, no máximo, o de índice *j-1*, utilizando o passo *k*.
- **lst[-k:]**: obtém os *k* últimos elementos da lista.
- **lst[:-k]**: em uma lista com *n* elementos, retornará os primeiros *n-k* elementos.

- Na Parte 4, mostramos que o operador **in** pode ser utilizado para testarmos se um determinado elemento pertence à lista (True ou False será retornado).
- Por fim, na Parte 5, apresenta-se o procedimento básico para modificar o conteúdo da lista. Veja que podemos modificar um único elemento, bastando indicar seu índice; ou vários de uma vez, utilizando a mesma notação empregada na operação de fatiamento.

Programa 26 – Outras duas operações permitidas sobre listas: repetição e concatenação.

```
#P026: repetição e concatenação de listas
lst1 = ['UFF', 'Niterói', 'RJ']
lst2 = ['Brasil']
```

```
#repetição: operador *
print(lst1*3) #['UFF', 'Niterói', 'RJ', 'UFF', 'Niterói', 'RJ', 'UFF', 'Niterói', 'RJ']

#concatenação: operador +
print(lst1 + lst2) #['UFF', 'Niterói', 'RJ', 'Brasil']
```

Saída [26]:

```
['UFF', 'Niterói', 'RJ', 'UFF', 'Niterói', 'RJ', 'UFF', 'Niterói', 'RJ']
['UFF', 'Niterói', 'RJ', 'Brasil']
```

19.3 Métodos

Além de ser uma ED, uma lista Python é também um **objeto**. Como o nome indica, objeto é um conceito relacionado ao paradigma da programação orientada a objetos, que foi brevemente mencionado na Lição 1 e que será abordado em mais detalhes no Capítulo IX desse livro.

Felizmente, não precisamos entender nada sobre programação orientada a objetos para **utilizar** os objetos da linguagem Python. Por enquanto, basta apenas saber que todo objeto é uma ED que contém **dados** (no caso de uma lista, os seus elementos) e **métodos** (funções presentes “dentro” do objeto e que sempre estão disponíveis para serem utilizadas pelo programador). Assim como ocorre com as funções normais (como `len()`, `type()`, etc.), um método pode receber zero ou mais argumentos e retornará um valor. A diferença é que, para chamar um método, devemos adicionar um ponto (“.”) e o nome do método ao final do nome de uma lista – na terminologia da programação orientada a objetos, isto é chamado de “invocar” o método (um nome um pouco feioso, mas não tão terrível quanto “indentação”). No Quadro 12 são relacionados alguns dos mais importantes métodos disponíveis para listas. Na notação adotada, considere que “lst” é uma lista e “e” um elemento.

Quadro 12. Métodos disponíveis para listas

- `lst.append(e)`: adiciona o elemento “e” à “lst”. Análogo ao comando `lst+=e`;
- `lst.clear()`: remove todos os elementos da lista;
- `lst.count(e)`: conta o número de ocorrências de “e” em “lst”;
- `lst.extend(lst2)`: insere a lista “lst2” no final de “lst”;
- `lst.index(e)`: retorna o menor índice de “e” em “lst”;
- `lst.insert(pos, e)`: insere o objeto “e” na posição “pos” de “lst”;
- `lst.remove(e)`: remove o elemento “e” de “lst”;
- `lst.reverse()`: inverte a lista;
- `lst.sort()`: ordena os elementos de “lst”.

Programa 27 – Exemplifica o uso dos métodos da ED lista.

```
#P027: métodos disponíveis para listas
numeros = [0, 10, 15, 10, 20]
print("lista original: ", numeros)

quantos_10 = numeros.count(10) #retorna 2
print("num. de ocorrências do valor 10: ", quantos_10)

i_10 = numeros.index(10) #retorna 1
```

```

print("índice da primeira ocorrência de 10: ", i_10)

numeros.append(5)      #adiciona o valor 5 no final da lista
print("lista modificada1: ", numeros)

numeros.insert(1,1000) #insere 1000 como segundo elemento
print("lista modificada4: ", numeros)

numeros.remove(10) #remove o primeiro elemento 10
print("lista modificada3: ", numeros)

numeros.extend([50, 60]) #adiciona a lista [50, 60] no final
print("lista modificada5: ", numeros)

numeros.sort() #ordena
print("lista ordenada: ", numeros)

numeros.reverse() #inverte
print("lista reversa: ", numeros)

numeros.clear() #esvazia a lista
print("lista vazia: ", numeros)

```

Saída [27]:

```

lista original: [0, 10, 15, 10, 20]
num. de ocorrências do valor 10: 2
índice da primeira ocorrência de 10: 1
lista modificada1: [0, 10, 15, 10, 20, 5]
lista modificada4: [0, 1000, 10, 15, 10, 20, 5]
lista modificada3: [0, 1000, 15, 10, 20, 5]
lista modificada5: [0, 1000, 15, 10, 20, 5, 50, 60]
lista ordenada: [0, 5, 10, 15, 20, 50, 60, 1000]
lista reversa: [1000, 60, 50, 20, 15, 10, 5, 0]
lista vazia: []

```

19.4 Funções Pré-Definidas e Listas

Algumas funções pré-definidas do Python podem ser diretamente aplicadas sobre listas, permitindo com que você execute certas operações sem a necessidade de criar laços. Elas são apresentadas no Quadro 13.

Quadro 13. Funções pré-definidas que podem ser usadas em listas

- **sum**(lst): soma os elementos da lista (apenas para listas com elementos numéricos);
- **min**(lst), **max**(lst): retornam, respectivamente o menor e o maior valor de "lst" (para qualquer tipo de lista);
- **len**(lst): retorna o tamanho da lista.

Programa 28 – Utilizando funções pré-definidas sobre listas. Abaixo são apresentados os resultados do teste de QI de oito participantes de um estudo.

Entrevistado	QI
Asif	126
Bill	100
Bob	130

Jim	92
Liu	120
Joan	99
Rakesh	125
Zangh	72

O programa a seguir mostra como utilizar funções pré-definidas para computar o QI máximo, mínimo e médio.

```
#P028: aplicando funções pré-definidas sobre listas
lst_qi = [126, 100, 130, 92, 120, 99, 125, 72]

print("resultados do teste de QI: ", lst_qi)
print("maior: ", max(lst_qi))
print("menor: ", min(lst_qi))
print("soma: ", sum(lst_qi))
print("media: ", sum(lst_qi) / len(lst_qi))
```

Saída [28]:

```
resultados do teste de QI: [125, 92, 72, 126, 120, 99, 130, 100]
maior: 130
menor: 72
soma: 864
media: 108.0
```

Listas e o módulo 'statistics'

- Não esqueça que todas as funções do módulo **'statistics'** (apresentado na Lição 15) também podem ser diretamente aplicadas sobre listas. Entre elas, funções para o cálculo da variância, desvio padrão, mediana, moda, etc.

19.5 Comparação de Listas

Os operadores de comparação podem ser utilizados em listas, tuplas (ver próxima seção), strings e outros tipos de sequências. A comparação começa pelo primeiro elemento de cada sequência. Se eles forem iguais, o segundo elemento será comparado e assim por diante, até que elementos diferentes sejam encontrados. Elementos subsequentes são ignorados. Exemplos:

```
[1,2,3] > [1, 5, 10]    #retorna False
[20] > [10, 998, 800]   #retorna True
```

19.6 Clonando uma Lista

Suponha que você tenha uma lista “a” e queira **gerar uma cópia** da mesma em uma outra lista “b”. Nesta situação, você **não** deve utilizar de jeito nenhum o comando abaixo:

- `b = a`

Como assim? O que ocorre é que, em Python, toda operação de atribuição que envolva objetos complexos (basicamente, tudo que não seja uma ED primitiva) não resultará na criação de um novo objeto; na verdade, a atribuição criará apenas uma nova variável que faz uma **referência** ao objeto original. No jargão da informática, esse processo é chamado de **cópia rasa** (*shallow copy*).

No entanto, em muitas situações práticas, você poderá estar interessado na operação conhecida como **cópia profunda** (*deep copy*), que consiste em criar uma cópia real ou um clone de um objeto. Neste caso, Python oferece duas saídas:

- **Fatiamento:** a operação de fatiamento de listas sempre gera clones (ou seja, as sublistas geradas são sempre cópias reais dos dados de uma lista). Desta forma, utilizando o comando `b = a[:]` você consegue obter uma sublista que na verdade é uma cópia completa da lista original.
- **Módulo 'copy':** oferece uma função chamada `deepcopy()` para realizar a cópia profunda não apenas de listas, mas de qualquer objeto complexo: `b = copy.deepcopy(a)`.

Programa 29 – Realizando a cópia rasa e a clonagem de listas.

```
#P029: shallow copy (referência) x deep copy (clonagem)
import copy

#1-Shallow Copy
print('* * SHALLOW COPY')
a = [1,2,3,4,5]
b = a
print('-a=',a)
print('-b=',b)

b[0] = 999
print('\t-a',a)
print('\t-b:',b)
print('\t-a is b?',a is b)          #True (o operador "is" verifica se dois objetos
                                   #      têm a mesma referência)
print('\n-----')

#2-Deep Copy utilizando a técnica de fatiamento
print('* * DEEP COPY COM FATIAMENTO')
c = [1,2,3,4,5]
d = c[:]
print('-c=',c)
print('-d=',d)
d[0] = 999
print('\t-c:',c)
print('\t-d:',d)
print('\t-c is d?',c is d)         #False
print('\n-----')

#3-Deep Copy utilizando o módulo 'copy'
print('* * DEEP COPY COM O MÓDULO \'copy\'')
e = copy.deepcopy(c)
print('-c=',c)
print('-e=',e)
e[0] = 999
print('\t-c:',c)
print('\t-e:',e)
print('\t-a is b?',c is d)         #False
```

Saída [29]:
* * SHALLOW COPY

```
-a= [1, 2, 3, 4, 5]
-b= [1, 2, 3, 4, 5]
  -a [999, 2, 3, 4, 5]
  -b: [999, 2, 3, 4, 5]
  -a is b? True
```

```
-----
* * DEEP COPY COM FATIAMENTO
```

```
-c= [1, 2, 3, 4, 5]
-d= [1, 2, 3, 4, 5]
  -c: [1, 2, 3, 4, 5]
  -d: [999, 2, 3, 4, 5]
  -c is d? False
```

```
-----
* * DEEP COPY COM O MÓDULO 'copy'
```

```
-c= [1, 2, 3, 4, 5]
-e= [1, 2, 3, 4, 5]
  -c: [1, 2, 3, 4, 5]
  -e: [999, 2, 3, 4, 5]
  -a is b? False
```

19.7 Listas como Argumentos de Funções

Na Lição 11, vimos que quando uma variável que armazena um valor primitivo (int, float, string ou boolean) é passada como argumento de uma função, ela é sempre um **parâmetro de valor**. Isto significa que qualquer alteração no conteúdo da variável que tenha sido realizada dentro do bloco de código da função, não será refletida para o programa principal. De maneira oposta, se um objeto complexo (como uma lista, dicionário, conjunto, etc.) é passado como argumento de uma função, ele será sempre um **parâmetro de referência**. Isto significa que alterações realizadas no formato ou conteúdo do objeto serão permanentes, ou seja, serão refletidas para o programa principal. Este comportamento é automático e não pode ser modificado (ao contrário do que ocorre em linguagens “raiz”, como C e Pascal).

Programa 30 – Passando um objeto de tipo complexo (lista) como argumento de função.

```
#P030: argumentos de função:
#      variável de tipo primitivo (int) x de tipo complexo (lista)
def f_dummy(z):
    if type(z) == list:
        z[0] = 999
    else:
        z = 999

x = 0
lst = [1,2,3,4,5]

print('x antes de chamar a função f_dummy:',x)
f_dummy(x)
print('x depois de chamar a função f_dummy:',x)

print('-----')

print('lst antes de chamar a função f_dummy:',lst)
f_dummy(lst)
print('lst depois de chamar a função f_dummy:',lst)
```

Saída [30]:

x antes de chamar a função f_dummy: 0
x depois de chamar a função f_dummy: 0

lst antes de chamar a função f_dummy: [1, 2, 3, 4, 5]
lst depois de chamar a função f_dummy: [999, 2, 3, 4, 5]

Parâmetros de Referência versus Parâmetros de Valor

- As alterações nos parâmetros de referência são permanentes porque, internamente, o Python passa para a função não uma cópia do objeto, mas o endereço do mesmo na memória. Isto é: o argumento da função e o objeto passado como parâmetro compartilham um mesmo endereço de memória. Portanto, uma alteração na primeira se reflete de forma permanente na segunda.
- Os parâmetros de valor funcionam de maneira inteiramente diferente: uma cópia temporária da variável passada como parâmetro é armazenada em alguma posição livre da memória. Durante a execução da função, somente essa cópia temporária é usada. Quando o valor do parâmetro é alterado, isso só tem efeito sobre o armazenamento temporário. A variável fora do procedimento jamais será tocada.



Lição 20 – Listas Bidimensionais

Uma estrutura essencial na matemática é a **matriz**, que nos permite representar dados retangulares dispostos em linhas e colunas. Na linguagem Python, qualquer matriz pode ser representada como uma **lista de listas**, ou seja, uma lista onde cada elemento também é uma lista. Para que o conceito fique claro, observe o exemplo abaixo, onde “m” é declarada como uma lista que possui como elementos outras duas listas, cada uma contendo três números inteiros como seus elementos. Isto é ou não uma matriz com 2 linhas e 3 colunas?

```
m = [  
    [1, 2, 3],  
    [4, 5, 6]  
]
```

O primeiro elemento de “m” – m[0] – é na verdade uma lista de números: [1, 2, 3]. O primeiro elemento dessa lista é m[0][0] é igual a 1. De maneira análoga, temos m[0][1]==2 e m[0][2]==3. O segundo elemento de “m” – m[1] – é a lista de números: [4, 5, 6] e temos que m[1][0]==4, m[1][1]==5 e m[1][2]==6.

As listas de listas são também chamadas de **listas bidimensionais** ou **listas 2d**. A seguir, veja mais dois exemplos de declaração de estruturas deste tipo:

```
matriz_binaria = [  
    [0, 1, 0, 0, 1, 0],  
    [1, 0, 0, 1, 1, 1],  
    [0, 0, 0, 0, 0, 1],  
    [0, 1, 1, 0, 1, 0]  
]
```

```
matriz_pesos = [
    [0.58, 0.19],
    [0.33, 0.65]
]
```

Curiosamente, as listas 2d não precisam necessariamente ser retangulares! Veja no exemplo abaixo a declaração de uma lista com 5 linhas, onde cada uma delas possui um número diferente de colunas:

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

Programa 31 – Operações Básicas sobre Listas Bidimensionais.

#P031: processamento básico de listas bidimensionais

```
m = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]
```

#(1)-indexando os elementos

```
print(m[0])      #retorna a primeira linha: [1, 2, 3, 4]
print(m[0][0])   #retorna 1 (elem da linha 0, col 0)
print(m[2][1])   #retorna 10 (elem da linha 2, col 1)
print(m[-1])     #retorna a última linha: [9, 10, 11, 12]
print(m[-2][3])  #retorna 8 (elem da penúltima linha, col 3)
print('-----')
```

#(2)-iteração: percorrendo os elementos com laços for-in aninhados

```
linhas = len(m)      #com len() obtemos o total de linhas
for y in range(linhas):
    cols = len(m[y])  #número de colunas da linha corrente
    for x in range(cols):
        print(m[y][x], end = " ")
    print();
print('-----')
```

#(3)-modificando o conteúdo

```
m[0][0] = 99        #modifica o elemento da linha 0, col 0 p/ 99
m[2] = [4, 3, 2, 1] #modifica a linha 2 p/ [4, 3, 2, 1]
print("nova matriz: ")
print(m)
```

#(3)-criando uma matriz dinamicamente

```
linhas=5
cols =2
m2 = []
for linha in range(linhas):
    m2 += [[0]*cols]
print("matriz criada dinamicamente: ")
```

```
print(m2)
```

Saída [31]:

```
[1, 2, 3, 4]
1
10
[9, 10, 11, 12]
8
-----
1 2 3 4
5 6 7 8
9 10 11 12
-----
nova matriz:
[[99, 2, 3, 4], [5, 6, 7, 8], [4, 3, 2, 1]]
-----
matriz criada dinamicamente:
[[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]
```

O programa está dividido em quatro partes:

- Na primeira, mostramos que para acessar qualquer elemento individual da lista 2d, nós precisamos utilizar colchetes duas vezes “[]” e especificar **primeiro o índice da linha e depois o índice da coluna**. Se utilizarmos colchetes apenas uma vez “[]”, uma linha inteira será recuperada.
- Na segunda parte, mostramos como utilizar duas estruturas **for...in** aninhadas para percorrermos as linhas e colunas da lista bidimensional, recuperando um elemento por vez. O laço externo percorre as linhas, enquanto o interno percorre as colunas. No exemplo, a função **len()** foi utilizada tanto para obtermos o número de linhas de “m” como para obtermos número de colunas de cada linha.
- Na terceira parte, mostrou-se como alterar um elemento e uma linha inteira da lista 2d.
- Por fim, a parte 4, que é a mais “original” do programa. Nela, mostramos como criar de forma dinâmica uma lista com 5 linhas e 2 colunas, onde todos os elementos possuem o valor 0. Primeiro criamos uma lista vazia (`m2=[]`) e depois utilizamos um laço **for...range**, para inserir cada linha. Dentro desse laço, cada iteração é responsável por gerar 2 colunas com valor 0 com o uso do operador de repetição “*”. Ainda dentro da iteração, essas colunas são adicionadas como uma nova linha de “m2” com o uso do operador de “+=” (atribuição com adição).

Infelizmente, algumas operações úteis não podem ser realizadas de forma direta sobre listas bidimensionais. Apenas para citar dois exemplos, não existe uma forma direta para selecionar uma coluna inteira ou para fatiar uma sub-matriz a partir da matriz original. Em ambos os casos, torna-se preciso programar laços aninhados e escrever algumas linhas de código para que seja possível obter os resultados desejados. De fato, na maioria das vezes que um programador Python deseja trabalhar com matrizes, ele opta por fazer uso da biblioteca ‘NumPy’ (*Numerical Python*), tema do Capítulo

VI do presente livro. Esta biblioteca estende a linguagem Python com a estrutura de dados “ndarray”, voltada para a computação de alto desempenho sobre vetores e matrizes.



Lição 21 – List Comprehension

List comprehension é uma **notação matemática** que facilita a criação e manipulação de listas. Para que você tenha ideia do quanto esse recurso é útil, suponha que você deseje criar uma lista contendo as potências de 2 variando do expoente 0 até o expoente 16. Ou seja, $lst = [2^0, 2^1, \dots, 2^{16}]$. A forma tradicional de fazer isso seria digitando todos os valores:

```
lst = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]
```

Entretanto, com o uso da notação *list comprehension* podemos declarar a mesma lista de uma forma muito mais elegante e compacta:

```
lst = [2**i for i in range(17)]
```

Literalmente: para “i” variando de 0 a 16, faça cada elemento da lista igual a 2^i . Muito prático não? Veja que realmente fica bem mais próximo da notação $lst = [2^0, 2^1, \dots, 2^{16}]$.

Generalizando, a definição de uma lista através da técnica *list comprehension* é feita com o uso da seguinte sintaxe:

```
lst = [x for x in sequência]
```

Veja mais alguns exemplos na tabela abaixo:

Notação Matemática	List Comprehension
$A = \{x^3 \mid 0 \leq x \leq 10\}$	<code>A = [x*x*x for x in range(11)]</code>
$B = \{1/2, 1/4, \dots, 1/10\}$	<code>B = [1/x for x in range(2,11,2)]</code>
$C = \{0, 0, 0, 0, 0, 0\}$	<code>C = [[([0]*2) for linha in range(5)]</code>
$D = \{x \mid 0 < x < 20 \text{ e } x \text{ é ímpar}\}$	<code>D = [x for x in range(20) if x%2==1]</code>

Os exemplos que criam as listas A e B são bem parecidos com o exemplo das potências de 2. No entanto, os dois últimos exemplos merecem uma explicação adicional:

- No terceiro exemplo, mostramos como utilizar a *list comprehension* como uma forma alternativa para criar uma lista bidimensional. A lista C criada armazena os dados de uma matriz com 5 linhas e 2 colunas, inteiramente preenchida com 0. Compare com o método usado no Programa 31 e veja como a nova solução é bem mais compacta.

- No quarto exemplo, utilizamos um teste condicional (`if x%2==1`) para fazer com que apenas os números ímpares fossem incluídos na lista. Essa mais uma funcionalidade interessante que podemos incorporar às nossas definições!



Lição 22 – Tuplas

Assim como as listas, as **tuplas** representam uma sequência de valores indexados através de números inteiros. No entanto, existe uma diferença crucial entre estas duas estruturas de dados: as tuplas são **imutáveis**, ou seja, não podem ter os seus elementos alterados. Por isso, elas podem ser utilizadas como chaves de dicionários (ver lição 25), entre outros tipos de papéis que não podem ser desempenhados por listas.

Programa 32 – Criação e manipulação básica de uma ED do tipo tupla. Veja que parênteses devem ser utilizados para criar a tupla.

```
#P032: tuplas - criação e manipulação básica

#tupla com 5 elementos
t1 = (10, 20, 30, 40, 50)

#para criar uma tupla com um único elemento,
#usa-se vírgula no final
t2 = (100,)

#tupla vazia - tuple() é o método construtor de tuplas.
t3 = tuple()

#se uma sequência é passada para tuple (ex: string ou lista),
#a tupla é criada com os elementos da sequência
t4 = tuple('DADOS')      #('D','A','D','O','S')

print('t1: ', t1)
print('tipo de dado: ', type(t1))
print('t1[0] -> ', t1[0])      #10
print('t1[2:] -> ', t1[2:])    #(30,40,50)
print('t2: ', t2)
print('t3: ', t3)
print('t4: ', t4)
```

Saída [32]:

```
t1: (10, 20, 30, 40, 50)
tipo de dado: <class 'tuple'>
t1[0] -> 10
t1[2:] -> (30, 40, 50)
t2: (100,)
t3: ()
t4: ('D', 'A', 'D', 'O', 'S')
```


Os operadores `*` (repetição), `+` (concatenação), `in` (teste de pertinência) também funcionam para tuplas, assim como as funções `len()`, `min()`, `max()` e `sum()`. No entanto, se você tentar alterar uma tupla obterá um erro, pois a tupla é imutável:

```
t1[0] = 1000
```

TypeError: 'tuple' object does not support item assignment

Tuplas versus Listas

- A diferença entre tuplas e listas é que tuplas são imutáveis, enquanto listas não. Isto significa que, uma vez definida, uma tupla não poderá ter valores inseridos, removidos ou modificados. A princípio, a "imutabilidade" das tuplas pode parecer algo sem sentido, mas na verdade trata-se de uma característica bastante útil em situações onde você precisa compartilhar dados com uma função ou um programa externo, mas não deseja que seus dados sejam modificados.
- Além disso, experimentos reportados em artigos envolvendo bases de dados reais e artificiais, evidenciaram que, para a maior parte das operações de busca, as tuplas são mais eficientes (rápidas) do que as listas.



Lição 23 – Tuple Assignment

Tuple assignment é funcionalidade muito interessante da linguagem Python que corresponde à capacidade de atribuir valores para mais de uma variável em uma única linha, utilizando uma estrutura do tipo tupla (veja o código abaixo). É um recurso muito utilizado em programas disponíveis em manuais e tutoriais, por isso é muito importante você o conheça!

```
a, b = ('John', 'Lennon')
print(a)
print(b)
```

Out[1]:
John
Lennon



Lição 24 – Conjuntos

Um conjunto é uma **coleção não-ordenada de elementos**. Ao contrário do que ocorre com as listas e tuplas, os conjuntos não podem ter elementos duplicados (isto é, um dado elemento aparecerá 0 ou 1 vez em um conjunto). Como não existe o conceito de ordenação em conjuntos, seus elementos não possuem índice.

Para criar um conjunto, basta especificar uma sequência de valores entre chaves, separando-os por vírgula. Veja alguns exemplos:

```
generos = {'Drama', 'Romance', 'Ação', 'Aventura'}  
numeros = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
conjunto_vazio = set()
```

Se tentarmos criar um conjunto com elementos repetidos, o Python removerá as repetições automaticamente:

```
x = {1, 1, 1, 1, 2}      #resulta em {1, 2}
```

A função `set()` também pode ser utilizada para possibilitar a criação de conjuntos a partir de sequências (listas, tuplas ou strings):

```
y = set("DADOS")        #resulta em {'D', 'A', 'O', 'S'}
```

Os conjuntos do Python têm o mesmo significado dos conjuntos matemáticos. Dessa forma, as principais operações sobre este tipo de ED são exatamente as tradicionais da matemática: união, interseção, diferença, relação de pertinência e relação de inclusão (o popular “contém / está contido”); além disso, existem métodos para incluir e remover elementos; por fim, também podemos iterar sobre os elementos de um conjunto com o comando **for**.

Programa 33 – Criação e manipulação básica de uma ED do tipo conjunto.

```
#P033: conjuntos - criação e manipulação básica
```

```
#define 3 conjuntos
```

```
a = {0,1,2,3}
```

```
b = {2,3,4,5}
```

```
c = {1,2}
```

```
#interseção (&), união (|), diferença (-) e diferença simétrica (^)
```

```
print (a & b)          #{2,3}
```

```
print (a | b)          #{0,1,2,3,4,5}
```

```
print (a - b)          #{0}
```

```
print (a ^ b)          #{0,1,4,5}
```

```
#está contido (<=), contém (>=)
```

```
print('-----')
```

```
print (c <= a)          #True
```

```
print (c <= b)          #False
```

```
print (a >= c)          #True
```

```
#pertence (in), não pertence (not in)
```

```
print('-----')
```

```
print (0 in a)          #True
```

```
print (0 in b)          #False
```

```
print (0 not in b)      #True
```

```
#incluindo (add) e removendo (remove) elementos;
```

```

print('-----')
c.add(10)
c.add(20)
print(c)           #{1,2,10,20}
c.remove(10)
print(c)           #{1,2,20}

#esvaziando um conjunto (clear) e fazendo deep copy (copy)
print('-----')
c.clear()
print(c)           #set()
d = a.copy()
print(d)           #(0,1,2,3)

#iteração
print('-----')
for item in b:
    print(item)

```

Saída [33]:

```

{2, 3}
{0, 1, 2, 3, 4, 5}
{0, 1}
{0, 1, 4, 5}

```

```

-----
True
False
True

```

```

-----
True
False
True

```

```

-----
{1, 2, 10, 20}
{1, 2, 20}

```

```

-----
set()
{0, 1, 2, 3}

```

```

-----
2
3
4
5

```



Lição 25 – Dicionários

25.1 Introdução

Um dicionário é uma ED em que os **elementos são pares chave:valor**. A chave (*key*) identifica um item e o valor (*value*) armazena o conteúdo do mesmo. Qualquer valor armazenado pode ser recuperado de forma extremamente rápida através de sua chave.

Uma das diferenças fundamentais entre os dicionários e as listas é que, em uma lista, os índices que determinam a posição dos elementos precisam ser inteiros, enquanto em um dicionário os índices podem ser não apenas inteiros, mas também de qualquer tipo básico imutável, como strings e tuplas. Outra diferença é que em um dicionário convencional¹⁶ não existe o conceito de ordem, ou seja, ele é uma coleção não ordenada de pares chave:valor. Veja alguns exemplos de declarações de dicionários:

```
dic_alunos = {'M01':'Jane', 'M13':'George', 'M15':'Thomas', 'M04':'Aldous'}

dic_titulos = {'Portela':22, 'Mangueira':19, 'Beija-Flor':14}

dicionario_vazio = {}
```

Observe que: (i) chaves "{ }" foram utilizadas para criar o dicionário; (ii) dentro das chaves, o símbolo de dois-pontos ":" é utilizado para separar a *key* de seu valor; e (iii) os diferentes elementos são separados por vírgulas.

25.2 Operações Básicas

Programa 34 – Operações de Inserção/Remoção de Itens, Busca e Modificação. O exemplo ilustra a criação e manipulação básica de um dicionário de alunos. Nele, a chave é a matrícula e o valor o nome do aluno. São apresentadas as operações básicas sobre dicionários: adicionar e remover itens, recuperar valores, modificar valores e pesquisar se uma chave pertence ao dicionário.

```
#P034: Dicionário: criação e manipulação básica
dic_alunos = {'M01':'Hane', 'M13':'George', 'M15':'Thomas'}
print('dic_alunos (original): ', dic_alunos)

#insere novo aluno (novo par chave:valor)
dic_alunos['M04'] = 'Aldous'

#altera o valor associado à chave 'M01' para Jane
dic_alunos['M01'] = 'Jane'

#remove o elemento de chave 'M15'
del dic_alunos['M15']

#checando se uma chave existe
tem_M13 = 'M13' in dic_alunos          #retorna True
tem_M99 = 'M99' in dic_alunos          #retorna False
print('dic_alunos (após alterações): ', dic_alunos)
print('tipo de dado: ', type(dic_alunos))
print('existe a mat. M13?: ', tem_M13)
print('existe a mat. M99?: ', tem_M99)
```

Saída [34]:

```
dic_alunos (original): {'M01': 'Hane', 'M13': 'George', 'M15': 'Thomas'}
dic_alunos (após alterações): {'M01': 'Jane', 'M13': 'George', 'M04': 'Aldous'}
tipo de dado: <class 'dict'>
existe a mat. M13?: True
existe a mat. M99?: False
```

16 Para trabalhar com dicionários ordenados é preciso utilizar outra ED. Um exemplo é a `OrderedDict`, que faz parte do módulo `'collections'`.

25.3 Métodos e Técnicas de Iteração

O Quadro 14 apresenta alguns dos principais métodos disponibilizados para dicionários. Na notação abaixo, considere que “dic” é um dicionário e “k” uma chave.

Quadro 14. Métodos disponíveis para dicionários

- `dic.keys()`: retorna uma referência para todas as chaves de “dic”;
- `dic.values()`: retorna uma referência para todos os valores de “dic”;
- `dic.items()`: retorna um objeto contendo todos os pares {chave:valor} de “dic”;
- `dic.clear()`: remove todos os elementos;
- `dic.get(k)`: recupera o valor do elemento de chave “k”. Se “k” não existir, retorna `None`;
- `dic.update(d)`: concatena o conteúdo de um dicionário “d” ao dicionário “dic”. Se algum par “chave:valor” de “d” já existir em “dic”, a informação será sobrescrita.

Programa 35 – Uso de métodos e Iteração sobre Dicionários. O Web site países do IBGE (<https://paises.ibge.gov.br/>) permite comparar países, através de seus principais indicadores demográficos, sociais, econômicos e ambientais. Dentre as diferentes informações disponibilizadas, encontra-se a população total de cada país. A relação abaixo, apresenta alguns exemplos:

Belize	179.014
Brasil	204.450.649
França	64.395.345
México	127.017.224
Nova Zelândia	2.213.123
Portugal	10.349.803
Uruguai	3.431.555

A seguir, apresenta-se um programa que cria um dicionário onde a chave é o nome de um país e o valor a sua população. O programa faz uso dos métodos descritos no Quadro 14, além de demonstrar como proceder para iterar sobre um dicionário, isto é obter todas as suas chaves e valores. Esta operação é realizada com o uso do método `items()` e do recurso de *tuple assignment*, apresentado na Lição 23.

```
#P035: uso de métodos, funções e iteração sobre um dicionário

#(0)-Cria o dicionário (por enquanto sem Belize e Nova Zelândia!)
d = {
    'Brasil': 204450649,
    'França': 64395345,
    'Portugal': 10349803,
    'México': 127017224,
    'Uruguai': 3431555,
}

#(1)-Utilizando os métodos
print('* * * 1-Métodos * * *')
print(d)
```

```

print(d.keys())
print(d.values())
print('A população estimada do Brasil é: ', d.get("Brasil"))

#(2)-percorre todos os elementos de "d"
# a cada iteração, a chave é armazenada em "k" e o valor em "v"
print('-----')
print('* * * 2-Percorrendo o dicionário * * *')
for k, v in d.items():
    print(k, '->', v)

#(3)-Utilizando as funções built-in:
# len(): conta o número de chaves armazenadas no dicionário
# min(): menor valor de chave
# max(): maior valor de chave
print('-----')
print('* * * 3-Usando funções built-in * * *')
print('Total de chaves: ', len(d))
print('menor chave: ', min(d))
print('maior chave: ', max(d))

#(4)-Combinando dois dicionários:
print('-----')
print('* * * 4-Combinando Dicionários * * *')
d2 = {
    'Belize': 179014,
    'Nova Zelândia': 2213123,
}

d.update(d2)
print('dicionário atualizado: ', d)

#(5)-removendo todos os itens do dicionário
print('-----')
print('* * * 5-Destruindo um dicionário * * *')
d.clear()
print(d)

```

Saída [35]:

```

* * * 1-Métodos * * *
{'Brasil': 204450649, 'França': 64395345, 'Uruguai': 3431555, 'Mexico': 127017224, 'Portugal': 10349803}
dict_keys(['Brasil', 'França', 'Uruguai', 'Mexico', 'Portugal'])
dict_values([204450649, 64395345, 3431555, 127017224, 10349803])
A população estimada do Brasil é: 204450649
-----
* * * 2-Percorrendo o dicionário * * *
Brasil -> 204450649
França -> 64395345
Uruguai -> 3431555
Mexico -> 127017224
Portugal -> 10349803
-----
* * * 3-Usando funções built-in * * *
Total de chaves: 5
menor chave: Brasil
maior chave: Uruguai

```

```
-----
* * * 4-Combinando Dicionários * * *
```

```
dicionário atualizado: {'Brasil': 204450649, 'França': 64395345, 'Uruguai': 3431555, 'Mexico':  
127017224, 'Portugal': 10349803, 'Belize': 179014, 'Nova Zelândia': 2213123}
```

```
-----
* * * 5-Destruindo um dicionário * * *
```

```
{}
```

25.4 **kwargs

Este recurso do Python permite com que você passe um dicionário qualquer como parâmetro de uma função. Isto pode ser feito com o uso de duas diferentes sintaxes, conforme ilustra o exemplo seguinte.

Programa 36 – função com número arbitrário de parâmetros estruturados em um dicionário

```
#P036: **kwargs
```

```
def pessoa(**kwargs):  
    for chave, valor in kwargs.items():  
        print("{0} = {1}".format(chave, valor))
```

```
#Sintaxe 1: nome=valor
```

```
pessoa(nome='John', profissao='musico')  
print("\n")
```

```
#Sintaxe 2: passando explicitamente um dicionário (**dicionario)
```

```
d={'nome':'Jane','profissao':'escritora','signo':'sagitário'}  
pessoa(**d)  
print("\n")
```

Saída [36]:

```
nome = John  
profissao = musico
```

```
nome = Jane  
profissao = escritora  
signo = sagitário
```

Veja que para obter todos os parâmetros dentro da função, é possível utilizar o super-útil método `items()`, que retorna todas os pares chave:valor do dicionário. Outra coisa interessante mostrada no programa é o uso da função `format()` dentro do comando `print()`. Esta função serve para formatar valores que serão colocados dentro do *placeholder* `{ }`. Trata-se de uma função muito utilizada pelos *pythonistas*.

Capítulo IV. Strings e Bases de Dados no Formato Texto

A maior parte da informação digital do planeta encontra-se disponibilizada no formato texto: blogs, páginas Web, livros digitais, redes sociais, wikis, e-mails, arquivos CSV, arquivos JSON, etc. Desta forma, não é nenhuma surpresa constatar que, nos anos recentes, a coleta, tratamento, sumarização e análise de texto digital tenham se tornado algumas das mais importantes tarefas de ciência de dados.

Tendo este cenário como motivação, este capítulo é dedicado à apresentação das técnicas básicas para processamento de texto oferecidas pela linguagem Python. O conteúdo está dividido em três partes:

- Parte 1 – Tratamento de strings. Introduz os diferentes métodos e funções para processamento de variáveis do tipo string.
- Parte 2 – Processamento de bases de dados no formato texto. Apresenta as técnicas que podem ser empregadas para que seu programa trabalhe com bases de dados textuais estruturadas em diferentes formatos e padrões de codificação.
- Parte 3 – Expressões regulares. Realiza uma breve introdução às expressões regulares, uma técnica que permite com que um padrão seja especificado (a tal expressão regular) e então utilizado para varrer um texto com o objetivo de extrair todos os trechos que “casem” com o padrão. Esta técnica é muito importante em processos de *web scraping* e mineração de texto.



Lição 26 – Strings

26.1 Introdução

Embora string seja um dos tipos primitivos do Python, qualquer variável com essa tipagem também pode ser manipulada como se fosse uma **tupla**. Essa é exatamente uma das características que torna a linguagem Python muito vantajosa para lidar com bases de dados textuais. Mais especificamente, uma string Python é considerada uma **sequência de caracteres**. Por exemplo, na string ‘Estatística’ (Figura 11), o caractere ‘E’ ocupa o índice 0, ‘s’ ocupa o índice 1 e assim por diante. É possível acessar cada caractere em separado, utilizando colchetes: “[]”.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
E	s	t	a	t	í	s	t	i	c	a

Figura 11. Uma string e seus índices

Programa 37 – Técnicas básicas para o processamento de strings. Veja que é igualzinho ao jeito com que trabalhamos com as listas e tuplas.

```
#P037: processamento básico de strings
palavra = 'Tiê Sangue'

#(1) indexando caracteres
prim_letra = palavra[0]           #retorna 'T'
ult_letra = palavra[len(palavra)-1] #retorna 'e'
tot_letras = len(palavra)         #retorna 10
print(prim_letra,ult_letra,tot_letras)
print('-----')

#(2) obtendo segmentos ou fatias (slices)
print(palavra[0:3])               #'Tiê'
print(palavra[4:11])              #'Sangue'
print(palavra[:3])                #'Tiê'
print(palavra[4:])                #'Sangue'
print(palavra[-1])                #'e' (última letra)
print(palavra[-2])                #'u' (penúltima letra)
print('-----')

#(3) percorrendo a string letra por letra, através de um laço
tot_esp = 0
for letra in palavra:
    print(letra)
    if letra == ' ':
        tot_esp = tot_esp + 1
print('"' + palavra + '"' possui ' + str(tot_esp) + ' espaço(s)')
print('-----')

#(4) operador "in"
tem_a = 'a' in palavra            #'Tiê Sangue' possui a letra 'a'?
tem_b = 'b' in palavra            #'Tiê Sangue' possui a letra 'b'?
print(tem_a, tem_b);
```

Saída [37]:

T e 10

Tiê
Sangue
Tiê
Sangue
e
u

T
i
ê

S
a
n
g
u
e

"Tiê Sangue" possui 1 espaço(s)

True False

Assim como ocorre com as tuplas, as strings são “imutáveis”, o que significa que não é possível alterar uma string existente. O máximo que se pode fazer é criar uma nova string que seja uma variação da original. Veja o exemplo abaixo:

Errado: bicho='Capivara' bicho[0] = 'K' TypeError: 'str' object does not support item assignment	Certo: bicho='Capivara' novo_bicho = 'K' + bicho[1:] print (novo_bicho) [1] Kapivara
--	--

26.2 Métodos

No Quadro 16 são relacionados alguns dos mais importantes métodos disponíveis para strings¹⁷. Na notação, considere que “s” é uma variável string qualquer.

17 Informações completas em: <https://docs.python.org/3.5/library/stdtypes.html#string-method>

Quadro 16. Métodos disponíveis para strings

- `s.lower()`: retorna uma cópia de "s" convertida para minúsculo; **Importante**: esse método, assim como todos os outros, **não** altera a string original "s", já que toda string é imutável!
- `s.upper()`: retorna uma cópia de "s" convertida para maiúsculo;
- `s.find(sub, inicio, fim)`: verifica se "sub" ocorre na string "s" ou dentro de algum trecho específico caso "início" e "fim" tenham sido definidos. Caso positivo, retorna o índice da primeira ocorrência. Senão retorna -1;
- `s.rfind(sub, inicio, fim)`: igual ao método `find()`, porém faz a verificação de trás para frente;
- `s.endswith(sufixo, inicio, fim)`: verifica se "s" ou um trecho definido de "s" (caso os parâmetros opcionais "início" e "fim" tenham sido especificados) termina com a substring especificada em "sufixo". Retorna `True` ou `False`;
- `s.replace(sub_ant, sub_nova, max)`: retorna uma cópia da string "s" com as ocorrências da substring "sub_ant" substituídas por "sub_nova". O parâmetro "max" pode ser utilizado para determinar o número máximo de trocas que serão realizadas.
- `s.count(sub, inicio, fim)`: conta quantas vezes a substring "sub" aparece no trecho de "s" demarcado por "início" e "fim" (se esses parâmetros forem omitidos procura na string toda).
- `s.strip()`: retorna uma cópia de "s" sem os espaços em branco à esquerda e à direita;
- `s.lstrip()`: retorna uma cópia de "s" sem os espaços em branco à esquerda;
- `s.rstrip()`: retorna uma cópia de "s" sem os espaços em branco à esquerda e à direita;
- `s.split(d,max)`: divide uma string em uma lista de strings, de acordo com o delimitador "d" (se este não tiver sido fornecido, usa espaço em branco). O parâmetro "max" pode ser utilizado para determinar o número máximo de elementos da lista.
- `s.translate(tabela, deletechars)`: este método é utilizado para traduzir strings de acordo com uma tabela de tradução (parâmetro "tabela") definida pelo método `maketrans()` (os dois métodos trabalham em conjunto). A sua utilização é um pouco complicada e nesse livro o utilizaremos apenas em sua forma básica, onde o parâmetro "deletechars" é utilizado para remover caracteres de uma string.

Programa 38 – Métodos disponíveis para strings. Uma observação sobre o programa abaixo: não existe um método para retornar o comprimento de uma string! Para obter esse valor, você precisa utilizar a velha e boa função pré-definida `len()`.

```
#P038: métodos de strings
p1 = 'Lagarto Teiú'

p1_maiusc = p1.upper()
p1_minusc = p1.lower()

num_letras_p1 = len(p1)
num_letras_a = p1.count('a')
testa_endswith = p1.endswith('rto',4,7)
testa_find = p1.find('a')
testa_rfind = p1.rfind('a')
```

```

p1_troca = p1.replace('a','o')
p1_split = p1.split()

p2 = '  Capivara ';
teste_strip = p2.strip();

p3 = 'ei, olha isso... uma capivara!'
sem_pontuacao = p3.translate(p3.maketrans('','','.,!")) #remove: , . !

print("p1.upper()= " + p1_maiusc)
print("p1.lower()= " + p1_minusc)
print("len(p1)= " + str(num_letras_p1))
print("p1.count('a')= " + str(num_letras_a))
print("p1.endswith('rto',4,7)= ",testa_endswith)
print("p1.find('a')= ",testa_find)
print("p1.rfind('a')= ",testa_rfind)
print("p1.replace('a','o')= " + p1_troca)
print("p1.split()= ", p1_split)
print("p2.strip()= *" + teste_strip + "*")
print("frase com pontuacao= " + p3)
print("frase sem pontuacao= " + sem_pontuacao)

```

Saída [38]:

```

p1.upper()= LAGARTO TEIÚ
p1.lower()= lagarto teiú
len(p1)= 12
p1.count('a')= 2
p1.endswith('rto',4,7)= True
p1.find('a')= 1
p1.rfind('a')= 3
p1.replace('a','o')= Logorto Teiú
p1.split()= ['Lagarto', 'Teiú']
p2.strip()= *Capivara*
frase com pontuacao= ei, olha isso... uma capivara!
frase sem pontuacao= ei olha isso uma capivara

```

Função dir()

- A função pré-definida `dir()` pode ser utilizada para listar todos os métodos disponíveis para um determinado objeto da linguagem Python. Veja a seguir que existem ainda muitos outros métodos disponíveis para strings, além daqueles que acabamos de apresentar.

```
palavra = 'Capivara'; print(dir(palavra))
```

Saída [1]:

```

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattr__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```

26.3 Comparação de Strings

Os operadores de comparação (`==`, `!=`, `<`, `<=`, `>`, `>=`) podem ser utilizados com strings. No entanto, é preciso estar atento às seguintes regras:

- Duas strings são iguais apenas se armazenam a mesma palavra, escrita de maneira idêntica, inclusive no que diz respeito às letras maiúsculas e minúsculas.
- As linguagens de programação não “agem” da mesma forma que as pessoas ao realizar comparações entre strings! Para as linguagens de programação, **letras maiúsculas vêm antes das letras minúsculas**. Isto ocorre porque as comparações entre strings são baseadas nos códigos internos dos caracteres e as letras maiúsculas possuem códigos menores do que as minúsculas. Por esse motivo, muitas vezes precisamos usar o método `lower()` (ou `upper()`) antes de comparar duas strings.
- De maneira análoga, **caracteres acentuados possuem códigos maiores do que letras minúsculas e números códigos menores do que qualquer letra**. Caso você deseje mais informações a este respeito, pesquise sobre “complete ascii table” e “unicode utf-8 table” na Internet.

Programa 39 – Comparação de strings.

```
#P039: comparação de strings
p1 = 'pássaro'
p2 = 'PÁSSARO'
p3 = '123'
print(p1=='pássaro')           #True
print(p1=='PáSSaRo')          #False
print(p1==p2)                  #False
print(p1.lower()==p2.lower())  #True
print(p3 < p1)                  #True

p1 = 'áaa'
p2 = 'aaa'
p3 = 'AAA'
print(p1==p2)                  #False
print(p2 < p1)                  #True
print(p2 < p3)                  #False
```

Saída [39]:

```
True
False
False
True
True
False
True
False
```

26.4 Módulo 'string'

O módulo 'string' é um antigo módulo da *standard library* que oferece uma série de constantes úteis para o tratamento de strings.

Programa 40 – Algumas constantes do módulo 'string'.

```
#P040: módulo string
import string
print(string.ascii_lowercase)    #abcdefghijklmnopqrstuvwxyz
print(string.ascii_uppercase)    #ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.ascii_letters)      #abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
                                #LMNOPQRSTUVWXYZ

print(string.digits)             #0123456789
print(string.punctuation)        #!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

Saída [40]:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```



Lição 27 – Arquivos Texto: File Handle

Chegou a hora! A partir dessa seção nossos programas começarão a trabalhar com **bases de dados**. Afinal de contas, analisar bases de dados é a razão de ser de qualquer cientista de dados!

Neste capítulo cobriremos as bases de dados no **formato texto**. Para trabalhar com bases deste tipo na linguagem Python, precisamos fazer uso da função **open()**, que permite a abertura de arquivos texto para leitura e escrita (gravação). Mas o que significa **abrir** um arquivo? Esta pergunta é, sem dúvida alguma, um tanto quanto importante! Por isso, merece uma resposta detalhada...

Toda vez que um programa Python precisa acessar um arquivo – seja para importá-lo por inteiro para a memória ou para processá-lo sequencialmente (linha por linha) – é necessário, antes de tudo, usar a função **open()** para comandar a sua abertura. Internamente (naqueles processos “misteriosos” que acontecem dentro do computador), a função **open()** estabelece uma “conversa” entre o Python e o sistema operacional do computador (Windows, Mac, Linux). Melhor explicando: abrir um arquivo, significa pedir ao sistema operacional para encontrar o endereço de localização do arquivo no HD, pen drive, cartão SD, enfim, no dispositivo em que ele esteja armazenado. Ao encontrar o endereço do arquivo, o sistema operacional retornará uma coisa chamada *file handle* para o programa Python. O *file handle* não é a mesma coisa que conteúdo do arquivo, ou seja, ele não consiste nos dados de fato. Na verdade, ele é uma espécie de ferramenta que permite ao programador “manejar” os dados do arquivo. Este conceito é representado no esquema¹⁸ da Figura 12.

Nas próximas lições, mostraremos como utilizar o modo básico oferecido pelo Python e pela sua *standard library* para trabalhar com arquivos texto. Mas, antes de qualquer coisa, é muito importante deixar claro que existem bibliotecas que oferecem maneiras bem mais práticas para trabalhar bases de dados texto, especialmente arquivos estruturados no formato CSV. Dois exemplos

18 Figura adaptada do livro “Python for Everybody: exploring data using python 3” de Charles R. Severance (2016).

são as bibliotecas ‘NumPy’ e ‘pandas’, respectivamente cobertas nos Capítulos VI e VII. De qualquer forma, consideramos importante que você primeiro conheça e aprenda a utilizar o jeito padrão do Python para lidar com arquivos por dois motivos:

1. A técnica baseada em *file handle* é, muitas vezes, a alternativa mais eficiente para realizar o **acesso sequencial** a arquivos, um processo onde apenas uma linha por vez é carregada para a memória. Em muitas aplicações reais, onde a base de dados a ser trabalhada é muito volumosa e não cabe em memória, o acesso sequencial pode ser o único método de acesso viável;
2. O *file handle* é simples e adequado para lidar com arquivos separados por colunas. Embora esse formato não seja muito popular nos dias de hoje – onde as pessoas tendem a adotar formatos bem mais bacanas como JSON e XML – ele ainda é utilizado para algumas aplicações específicas. Por exemplo, é adotado por muitos países para armazenar dados de censos demográficos.

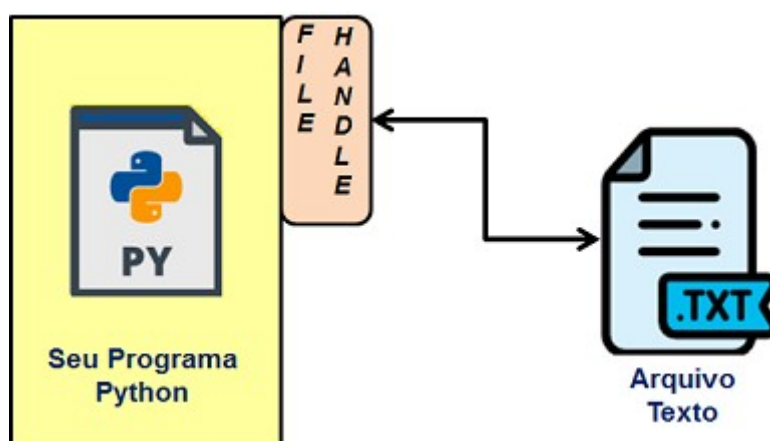


Figura 12. *File handle*



Lição 28 – Arquivos Texto: Processando Arquivos Separado por Colunas

Programa 41 – Leitura de Arquivo Separado por Colunas – Tamanho Fixo. O arquivo “ARQ_COLUNAS.txt”, apresentado abaixo, armazena informações de duas variáveis: “n1”, uma variável numérica, que inicia na coluna 1 e possui tamanho fixo de 4 colunas; e “c1”, uma variável do tipo caractere, iniciando na coluna 5 e com tamanho fixo de 5 colunas.

```
1001aaaaa
1002bbbbb
1003ccccc
1004ddddd
1005eeeeee
```

Suponha que este arquivo esteja armazenado na pasta “C:\CursoPython” (utilizaremos essa suposição para todos os exemplos de agora em diante – modifique o código se você estiver

utilizando outra pasta). A seguir, apresenta-se um programa que abre o arquivo, realiza a leitura linha por linha (acesso sequencial) e captura as duas variáveis.

#P041: leitura de arquivo separado por colunas

```
nomeArq = 'C:/CursoPython/ARQ_COLUNAS.txt'
f = open(nomeArq)

for linha in f:
    n1 = linha[:4]
    c1 = linha[4:]
    print(n1, c1)
```

Saída [41]:

1001 aaaaa

1002 bbbbbb

1003 ccccc

1004 ddddd

1005 eeeee

Agora uma explicação sobre o funcionamento do programa. Inicialmente, a função `open()` é utilizada para abrir o arquivo, cujo caminho está armazenado na variável string “nomeArq”. A variável “f”, que recebe o resultado retornado pela função `open()`, não armazena o arquivo propriamente dito, mas sim um *file handle* retornado pelo sistema operacional, que representa uma espécie de “ponteiro” para o arquivo real (aponta para o endereço do arquivo). Por isso, se você mandar imprimir “f”, não verá o conteúdo do arquivo na tela, mas sim um texto esquisito que descreve o *file handle*.

O comando `for linha in f:` possibilita com que o arquivo apontado por “f” seja varrido da primeira à última linha. Em outras palavras, este comando implementa o acesso sequencial ao arquivo. Nesta técnica, o arquivo não é importado para a memória. Ao contrário, a cada iteração do `for`, apenas uma linha é carregada para a memória, tendo o seu conteúdo copiado para a variável “linha”. Dentro de cada iteração, o comando `n1 = linha[:4]` faz com que a substring formada pelos 4 primeiros caracteres de “linha” seja armazenada na variável “n1”. Lembre-se de que o primeiro caractere de uma string sempre possui o índice 0 – sendo assim, o comando apresentado captura os caracteres de índice 0, 1, 2 e 3. De maneira análoga, `c1 = linha[4:]` copia para “c1” a substring que começa no índice 4 e vai até o último caractere de “linha”.

Ao executar o exemplo, você deve ter percebido que uma linha em branco foi impressa entre cada resultado produzido por `print(n1, c1)`. Isto ocorreu porque o **último caractere de uma linha lida do arquivo é na verdade o caractere “\n”** (ele é invisível para nós, mas existe no arquivo!). Para evitar a impressão do “\n” (o cara que gera linha em branco), basta efetuar a seguinte modificação no programa:

```
c1 = linha[4:9]
```

Desta maneira, “c1” receberá a substring compreendida entre o quarto e o oitavo caractere da variável “linha” (não esqueça que o fatiamento sempre vai do primeiro índice até o último menos um). Outra forma possível é, logo abaixo da instrução `for linha in f:`, acrescentar uma nova

linha de comando: `linha = linha.strip()`. Esta solução funciona porque o método `strip()` remove caracteres especiais no início e no fim de uma string.

Outra coisa importante de ser mencionada é que no acesso sequencial a gente só consegue “andar pra frente” no arquivo. Uma vez que uma linha seja alcançada, não há como voltar para a linha anterior ... a não ser que você feche e abra o arquivo novamente, e implemente outro comando **for** para varrer o arquivo até chegar na linha desejada.

Programa 42 – Leitura de Arquivo Separado por Colunas – Tamanho Variável. Considere agora o arquivo “PRODUTOS.txt”, que contém o código e o nome de alguns produtos. Apesar de se tratar de um arquivo separado por colunas, ele é diferente do arquivo mostrado na subseção anterior, pois o comprimento de cada uma de suas linhas é variável (os nomes de cada produto possuem comprimento diferente).

```
1001Leite
1002Biscoito
1003Café
1004Torradas
1005Chá
```

Para que o arquivo seja processado corretamente, o macete é utilizar a função `len()`, que retorna o comprimento de uma string. Entretanto, tome **cuidado**: pois o valor retornado inclui o chato do “\n”, uma vez que estamos lidando com linhas de arquivo e esse cara está escondidinho no final de cada linha.

```
#P042 leitura de arquivo separado por colunas - tamanho variável
nomeArq = 'C:/CursoPython/PRODUTOS.txt'
f = open(nomeArq)

for linha in f:
    codigo = linha[:4]
    nome = linha[4:len(linha)-1]
    print(codigo, nome)
```

Saída [42]:

```
1001 Leite
1002 Biscoito
1003 Café
1004 Torradas
1005 Chá
```



Lição 29 – Arquivos Texto: Importando um Arquivo Inteiro para uma String

Caso você esteja trabalhando com um arquivo não muito volumoso, poderá importá-lo inteiramente para a memória através do método `read()`, que é oferecido pelo objeto *file handle* (sim, sim o *file handle* é um objeto Python!). Quando um arquivo é lido dessa forma, todos os seus caracteres (incluindo todos os sem vergonhas dos “\n”!) são armazenados em um único “stringão”,

como mostra o exemplo a seguir. Obviamente, esse processo não pode ser realizado para arquivos muito grandes.

Programa 43 – Leitura de um Arquivo Inteiro para uma String

```
#P043: importa arquivo inteiro para um "stringão"
nomeArq = 'C:/CursoPython/PRODUTOS.txt'
f = open(nomeArq)
conteudo = f.read()
print(conteudo)
```

Saída [43]:

```
1001Leite
1002Biscoito
1003Café
1004Torradas
1005Chá
```



Lição 30 – Arquivos Texto: Processando Arquivos CSV

Com o uso do acesso sequencial podemos manipular arquivos CSV (*comma-separated values* – valores separados por vírgula) e outros tipos de arquivos baseados em caracteres delimitadores. Veremos como fazer isso nos exemplos desta lição.

Programa 44 – Leitura de Arquivo CSV - Processo Básico. O arquivo “CEPS.csv” armazena informações sobre faixas de CEPs utilizadas em estados da região Sudeste. O primeiro valor corresponde ao CEP inicial, o segundo é o CEP final e o último o nome da UF. A primeira linha do arquivo contém o cabeçalho, ou seja, a descrição das variáveis.

```
cep_ini, cep_fim, nome_uf
20000000,28999999,Rio de Janeiro
29000000,29999999,Espírito Santo
30000000,39999999,Minas Gerais
01000000,19999999,São Paulo
```

Para realizar leitura deste arquivo CSV separando as variáveis de forma correta, podemos fazer uso do método **split()**, que é disponibilizado automaticamente para qualquer variável do tipo string. Este método quebra uma string em uma lista de palavras, bastando indicar o caractere delimitador como parâmetro. Um exemplo é apresentado no código a seguir, que abre o arquivo “CEPS.csv” para acesso sequencial, “pula” a linha de cabeçalho e imprime o CEP inicial e o CEP final de cada UF.

```
#P044: leitura de arquivo separado por delimitador
nomeArq = 'C:/CursoPython/CEPS.csv'
f = open(nomeArq)

aux=0 #auxiliar para permitir que cabeçalho seja ignorado
for linha in f:
    if (aux > 0): #ignora a linha de cabeçalho
```

```

linha = linha[:len(linha)-1] #remove o tremendamente chato do "\n"
lstPalavras = linha.split(",")
cep_ini = lstPalavras[0]
cep_fim = lstPalavras[1]
uf = lstPalavras[2]
print(uf + " -> CEPS de " + cep_ini + " a " + cep_fim)
aux=aux+1

```

Saída [44]:

Rio de Janeiro -> CEPS de 20000000 a 28999999

Espírito Santo -> CEPS de 29000000 a 29999999

Minas Gerais -> CEPS de 30000000 a 39999999

São Paulo -> CEPS de 01000000 a 19999999

Programa 45 – Conversão de Tipos. O arquivo “FUNCIONARIOS.csv”, apresentado a seguir, possui o nome, idade e salário dos funcionários de uma empresa hipotética.

```

Jane;55;2500.00
Gregory;32;1200.50
Rakesh;48;4999.99
Mia;29;1900.00
Pete;50;2900.00

```

Considere que a empresa deseja dar um prêmio equivalente a 15% do salário para todos os funcionários com idade igual ou superior a 50 anos. Para elaborar um programa capaz de identificar tais funcionários e ainda determinar o valor do bônus a ser pago a cada um deles, torna-se necessário implementar duas coisas. Primeiro, devemos ler cada linha do arquivo, separando as informações três variáveis distintas, “nome”, “idade” e “salario”. Segundo, precisamos especificar que a variável “idade” é do tipo int (já que precisaremos realizar uma comparação envolvendo a idade) e “salário” do tipo float (pois precisaremos realizar um cálculo baseado no valor do salário). Como vimos no Capítulo II, a linguagem Python oferece funções pré-definidas para conversão de tipo que podem ser utilizadas em situações como essa:

- `int(x)`: converte “x” para um int, onde “x” pode ser uma string ou um float;
- `float(x)`: converte “x” para um float, onde “x” pode ser uma string ou um int.
- `str(x)`: converte “x” para uma string, onde “x” pode ser um int ou um float.

O código a seguir resolve o problema proposto. Analise-o com cuidado:

```

#P045: leitura de arquivo CSV e conversão de tipos
nomeArq = 'C:/CursoPython/FUNCIONARIOS.csv'
f = open(nomeArq)

for linha in f:
    linha = linha[:len(linha)-1] #remove o chato do "\n"
    lstAux = linha.split(";")
    nome = lstAux[0]
    idade = int(lstAux[1])
    salario = float(lstAux[2])
    if (idade > 49):
        premio = salario * 0.15

```

```
print(nome + " -> ganhou um premio de R$ " + str(premio))
```

Saída [45]:

Jane -> ganhou um premio de R\$ 375.0

Pete -> ganhou um premio de R\$ 435.0



Lição 31 – Arquivos Texto: Gravando Arquivos

Para gravar um arquivo, você deve abri-lo utilizando o modo “w” como segundo parâmetro da função `open()`. Quando a gravação terminar, é necessário utilizar a função `close()` para fechar o arquivo.

Programa 46 – Gravação de Arquivos Texto.

```
#P046 - Gravando um arquivo texto
fout = open('C:/CursoPython/toque_fragil.txt', 'w')
msg1 = "O sorriso\n"
msg2 = "Do\n"
msg3 = "Cachorro\n"
msg4 = "Tá\n"
msg5 = "No rabo...\n"

fout.write("Toque Frágil (Walter Franco)\n")
fout.write("===== ")
fout.write("\n")
fout.write(msg1 + msg2 + msg3 + msg4 + msg5)
fout.close()
```

Após a execução, o programa terá gerado um arquivo com seis linhas, chamado “toque_fragil.txt” na pasta “C:\CursoPython” (Fig. 13).

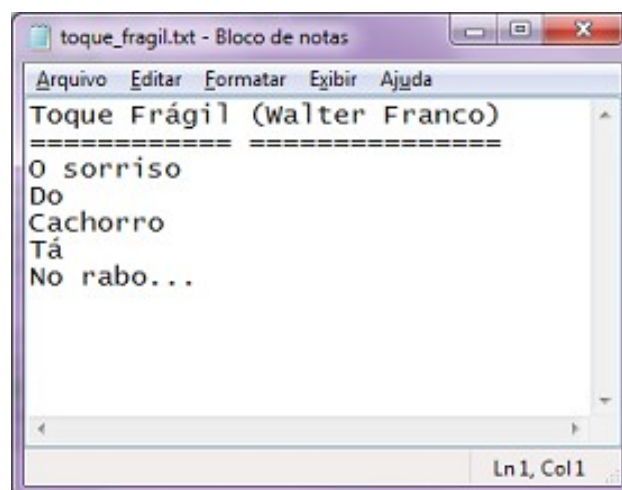


Figura 13. Arquivo gravado

❗ Outros modos para abertura de arquivos com o comando `open()`

- Além dos modos de leitura ('r') e escrita ('w'), o comando `open()` suporta também os seguintes modos:
 - **modo append** ('a'): quando a intenção é apenas adicionar linhas ao final de um arquivo existente, esse é o modo adequado. Em uma situação dessas, não vale apenas usar o modo 'w', pois qualquer arquivo existente que seja aberto no modo 'w' tem o seu conteúdo totalmente apagado.
 - **modo binário** ('b'): utilizado quando precisamos lidar não com arquivos texto, mas sim com arquivos binários. Por exemplo, para abrir uma imagem GIF contendo o logotipo de uma empresa no modo binário, devemos usar `logotipo = open('logo.gif', 'rb')`.



Lição 32 – Arquivos Texto: Conhecendo o Padrão Unicode

32.1 O que é Unicode?

Unicode é um **padrão** adotado mundialmente que possibilita com que **todos os caracteres** de **todas as linguagens** escritas utilizadas no planeta possam ser representados em computadores. A missão do Unicode é apresentada de forma clara no web site do *Unicode Consortium*¹⁹ (entidade responsável pela sua gestão):

*Unicode fornece um número único para cada caractere,
não importa a plataforma,
não importa o programa,
não importa a linguagem.*

O padrão Unicode é capaz de representar não somente as letras utilizadas pelas linguagens mais familiares para nós ocidentais, como Inglês, Espanhol, Francês e o nosso Português, mas também letras e símbolos utilizados em qualquer outra linguagem: Russo, Japonês, Chinês, Hebreu, etc. Além disso, inclui símbolos de pontuação, símbolos técnicos e outros caracteres que podem ser utilizados em texto escrito.

32.2. Como o Unicode Trabalha?

No padrão Unicode, cada diferente letra ou símbolo de cada alfabeto utilizado no mundo é mapeado para um diferente **code point**. O *code point* é um código no formato **U + número em hexadecimal**. O exemplo abaixo mostra os códigos das letras que compõem a palavra “BRASIL” (em maiúsculo).

B - U+0042
R - U+0052
A - U+0041
S - U+0053

¹⁹ <http://unicode.org/consortium/consort.html>

I - U+0049
L - U+004C

É muito importante mencionar que as letras maiúsculas possuem *code points* diferentes das letras minúsculas. Por exemplo: o *code point* da letra “A” é U+0041, enquanto o da letra “a” é U+0061, o *code point* de “Ç” é U+00C7 e o de “ç” é U+00E7 (e por aí vai). Outra observação importante é que os primeiros 127 *code points* (até U+007F) são compatíveis com os códigos utilizados na antiga tabela ASCII, a primeira criada com objetivo de padronizar a codificação dos caracteres. Estes 127 *code points* representam, basicamente, os códigos associados aos números, letras maiúsculas e minúsculas sem acento e símbolos de pontuação mais comuns.

O aplicativo **Mapa de caracteres** (*charmap*) do Windows pode ser utilizado para consulta à tabela Unicode. Para acessá-lo, basta ir para o Prompt de Comando e digitar `charmap`. Na Figura 14, o Mapa de caracteres informa o *code point* associado à letra Á (“A” maiúsculo com acento agudo).

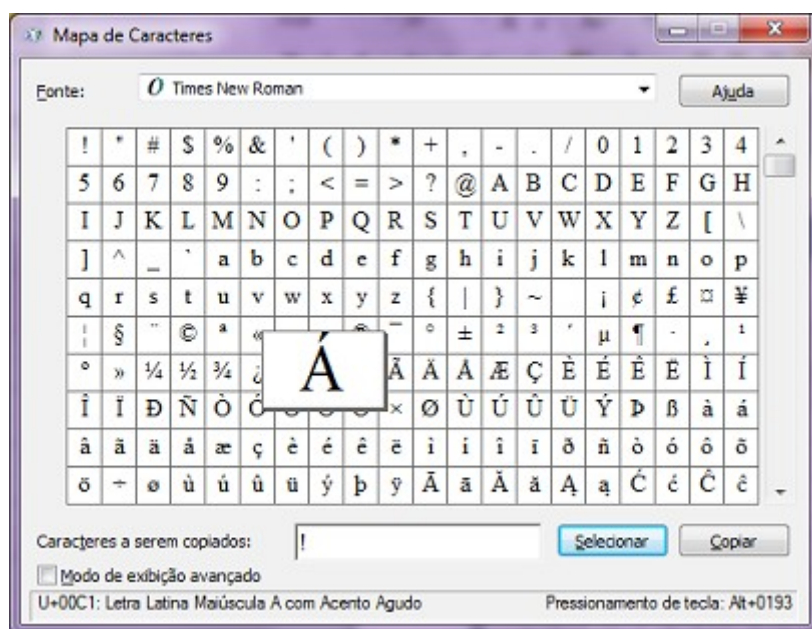


Figura 14. Mapa de caracteres

32.3. Encodings

Do que foi apresentado até agora, podemos entender que o Unicode nada mais é do que uma **enorme tabela** que associa um número único (*code point*) para cada diferente letra ou símbolo dos alfabetos de todo o mundo. Mas como esses *code points* podem ser armazenados em um arquivo texto ou na memória do computador? É aí que entram em cena os **encodings**.

Um *encoding* é um esquema de armazenamento dos *code points* dos caracteres que compõem as strings na memória do computador. Existem vários deles: **UTF-8**, ISO-8859-1 (apelidado de Latin-1), UCS-2, **ANSI** (ou Windows-1252), etc.

Cada *encoding* utiliza uma técnica distinta para lidar com os códigos Unicode²⁰. O *encoding* UTF-8, por exemplo, é capaz de representar qualquer caractere Unicode. Para conseguir isso, utiliza

²⁰ Para aprender mais sobre o assunto, consulte o tutorial disponibilizado em: <http://kunststube.net/encoding/>

uma técnica onde uma quantidade de 1 a 6 bytes pode ser utilizada para representar cada caractere. Ou seja, o UTF-8 não trabalha com uma representação em tamanho fixo. Os *code points* de 0 a 127 são armazenados com 1 byte. Porém, os demais podem ser armazenados em memória com tamanho de 2 a 6 bytes. O UTF-8 é completo (armazena qualquer caractere Unicode) e usa uma técnica de armazenamento que pode ser considerada “sofisticada”. Outros *encodings*, como ISO-8859-1 e ANSI são menos completos do que o UTF-8, e preferem trabalhar apenas com um subconjunto da tabela Unicode (ex: apenas com caracteres das linguagens ocidentais). Em compensação, podem fazer uso de técnicas de armazenamento mais simples e que conseguem representar strings gastando um número menor de bytes.

Mas por que é tão importante que um cientista de dados saiba o que é Unicode, *encoding*, padrão UTF-8, padrão Latin-1, padrão ANSI, etc., etc.? É porque em muitas situações práticas, torna-se necessário avisar ao Python (na verdade, a qualquer linguagem de programação) a codificação de um arquivo para que seja possível abri-lo. Melhor detalhando: quando um arquivo contém apenas os caracteres convencionais (*code points* de 0 a 127, ou seja, letras sem acento, números, sinais de pontuação mais comuns, tab, espaço em branco, “\n”, etc.) não importa a sua codificação, pois todos os *encodings* utilizam o mesmo esquema interno para armazená-los. Mas se o arquivo contém algum caractere especial, mesmo que seja uma inocente cedilha ou uma letra “a” acentuada, o Python não conseguirá processá-lo se você não informar o *encoding* correto, pois os diferentes *encodings* usam técnicas distintas para armazenar caracteres especiais. Não se preocupe, porque mostraremos com isso funciona na prática ainda neste capítulo, mais especificamente na Lição 34.



Lição 33 – Arquivos Texto: Módulo ‘csv’

Na Lição 30, mostramos que é possível acessar arquivos CSV utilizando a função `open()` em conjunto com o método de string `split()`. No entanto, a *standard library* oferece o módulo ‘csv’, que tem a interessante capacidade de associar linhas de arquivos CSV a listas e dicionários.

Programa 47 – Módulo ‘csv’ no Modo Lista. A utilização do módulo ‘csv’ será demonstrada através de exemplos que utilizam o arquivo “PAISES10.csv”. Este arquivo armazena uma pequena base de dados que possui 10 observações e 5 variáveis: sigla do país, nome do país, continente (‘A’=América ou ‘E’=Europa), extensão territorial (em km²) e tamanho da população.

```
sigla;nome;continente;extensao;populacao
BRA;Brasil;A;8515767;204450649
CUB;Cuba;A;109890;11389562
FRA;França;E;549190;64395345
HUN;Hungria;E;93030;9855023
ITA;Itália;E;301340;59797685
MEX;México;A;1964380;127017224
NOR;Noruega;E;323780;5210967
PER;Peru;A;1285220;31376670
PRT;Portugal;E;92090;10349803
URY;Uruguai;A;176220;3431555
```

```
#P047: trabalhando com o módulo "csv" no modo LISTA
import csv
```



```
with open('C:/CursoPython/paises10.csv','rt') as f:
    meu_csv = csv.reader(f, delimiter=';')
    for linha in meu_csv:
        print(linha)
```

Saída [47]:

```
['sigla', 'nome', 'continente', 'extensao', 'populacao']
['BRA', 'Brasil', 'A', '8515767', '204450649']
['CUB', 'Cuba', 'A', '109890', '11389562']
['FRA', 'França', 'E', '549190', '64395345']
['HUN', 'Hungria', 'E', '93030', '9855023']
['ITA', 'Itália', 'E', '301340', '59797685']
['MEX', 'México', 'A', '1964380', '127017224']
['NOR', 'Noruega', 'E', '323780', '5210967']
['PER', 'Peru', 'A', '1285220', '31376670']
['PRT', 'Portugal', 'E', '92090', '10349803']
['URY', 'Uruguai', 'A', '176220', '3431555']
```

O programa possui apenas 5 linhas, que são explicadas a seguir:

- **import csv :**
 - Importa o módulo ‘csv’.
- **with open('C:/CursoPython/paises10csv','rt') as f:**
 - Abre o arquivo para leitura (parâmetro ‘rt’), associando-o ao *file handle* “f”.
- **meu_csv = csv.reader(f, delimiter=';')**
 - Cria o objeto “meu_csv”, do tipo `csv.reader`. Esse tipo de objeto “entende” o que é um arquivo CSV e permite a iteração sobre as linhas do mesmo. Veja que dois parâmetros foram passados: o file handle (“f”) e o delimitador (“;”).
- **for linha in meu_csv:**
 - print(linha)**
 - Varre todo o arquivo CSV de forma sequencial, da primeira à última linha.

Comando with

- O comando `with` permite com que arquivos possam ser abertos com a função `open()` sem a necessidade de utilizar a função `close()` para fechá-los. Isso porque o `with` se encarrega de fechar automaticamente o arquivo, mesmo que ocorra um erro no meio da execução do programa. Devido a esta vantagem, muitos pacotes do Python exigem a utilização do `with` para abrir arquivos.

Programa 48 – Escolhendo as Variáveis de Interesse. Com o uso do pacote “csv”, cada linha do arquivo é carregada como uma lista em vez de uma string. Isto facilita bastante o processamento do arquivo, pois cada variável fica associada a um índice específico. Em nosso exemplo, “sigla” fica com o índice 0, “nome” com o índice 1, etc. O programa abaixo imprime apenas a sigla e a população de cada país, além de não imprimir o cabeçalho.


```
#P048: lê o CSV e imprime apenas 2 variáveis
import csv
with open('C:/CursoPython/paises10.csv','rt') as f:
    meu_csv = csv.reader(f, delimiter=';')
    i=0;
    for linha in meu_csv:
        if i > 0: #para ignorar o cabeçalho
            print(linha[0] + ' -> população = ' + linha[4])
            i = i+1
```

Saída [48]:

```
BRA -> população = 204450649
CUB -> população = 11389562
FRA -> população = 64395345
HUN -> população = 9855023
ITA -> população = 59797685
MEX -> população = 127017224
NOR -> população = 5210967
PER -> população = 31376670
PRT -> população = 10349803
URY -> população = 3431555
```

Programa 49 – Módulo 'csv' no Modo Dicionário. Com o uso do método `DictReader()` é possível estruturar cada linha de um arquivo CSV em um dicionário em vez de uma lista. Com o uso deste módulo: (i) a linha de cabeçalho é interpretada e importada de maneira automática; (ii) cada variável pode ser referenciada pelo seu nome e não pela sua posição.

```
#P049: lendo um arquivo CSV como um dicionário
import csv
with open('C:/CursoPython/paises10.csv','rt') as f:
    meu_csv = csv.DictReader(f, delimiter=';')
    for linha in meu_csv:
        print(linha["sigla"] + ' -> populacao = ' +
              linha["populacao"])
```

Saída [49]:

```
BRA -> população = 204450649
CUB -> população = 11389562
FRA -> população = 64395345
HUN -> população = 9855023
ITA -> população = 59797685
MEX -> população = 127017224
NOR -> população = 5210967
PER -> população = 31376670
PRT -> população = 10349803
URY -> população = 3431555
```

Módulo 'csv'

- O módulo 'csv' oferece diversas outras funcionalidades avançadas para a interpretação de arquivos CSV (ex.: dialetos, parâmetros de formatação, etc.). No entanto, como nosso livro é introdutório, preferimos mostrar apenas a "receita de bolo".

- Adicionalmente, é importante mencionar que em capítulos posteriores mostraremos como as bibliotecas 'NumPy' e 'pandas' podem ser utilizadas para realizar a importação de arquivos CSV diretamente para vetores, matrizes e DataFrames em memória. Na prática, essas duas bibliotecas são muito mais utilizadas para trabalhar com arquivos CSV do que o módulo 'csv'.



Lição 34 – Arquivos Texto: UTF-8 versus ANSI

Conforme previamente introduzido na Lição 32, os *encodings* costumam dar muita dor de cabeça para não-informatas que estão desenvolvendo *scripts* em Python, R e outras linguagens. Recordando, um *encoding* consiste em um esquema utilizado para o armazenamento de códigos de caracteres. Os diferentes *encodings*, como UTF-8 e ANSI, utilizam técnicas de armazenamento distintas e, por este motivo, ao trabalhar com um arquivo texto em um programa Python, é importante identificar o *encoding* do mesmo para que não ocorra erro no processamento do programa.

Programa 50 – Arquivo com Codificação UTF-8 (abrindo do jeito errado). Suponha que o arquivo “ARTILHEIROS.csv”, listado a seguir, foi gravado com a codificação UTF-8. Veja que, aparentemente, ele não possui nenhuma diferença em relação a qualquer outro arquivo:

```
jogador,gols
Hélio,20
Andrey,15
João,12
Antônio,11
```

No entanto, se você tentar abrir este arquivo com o comando `open()` sem especificar o *encoding* UTF-8 poderá ocorrer um erro ou, pior, o seu programa pode gerar resultados “malucos”, como mostra o exemplo abaixo:

```
#P050: PROBLEMA ao abrir arquivo na codificação utf-8
import csv
with open('C:/CursoPython/ARTILHEIROS.csv','rt') as f:
    meu_csv = csv.reader(f, delimiter=',')
    for linha in meu_csv:
        print(linha[0] + ' -> ' + linha[1])
```

Saída [50]:

```
ï»¿jogador -> gols
HÃ©lio -> 20
Andrey -> 15
JoÃ£o -> 12
AntÃ´nio -> 11
```

Veja que os caracteres acentuados presentes nos nomes não foram impressos corretamente e ainda apareceram uns símbolos estranhos antes da palavra “jogador”. Felizmente, a correção do

problema é simples: o parâmetro **encoding** deve ser utilizado no comando **open()**, para indicar que o arquivo está codificado no formato “utf-8”

Programa 51 – Arquivo com Codificação UTF-8 (abrindo do jeito certo).

```
#P051: FORMA CERTA de abrir arquivo na codificação utf-8
import csv
with open('C:/CursoPython/ARTILHEIROS.csv','rt', encoding="utf-8") as f:
    meu_csv = csv.reader(f, delimiter=',')
    for linha in meu_csv:
        print(linha[0] + ' -> ' + linha[1])
```

Saída [51]:

```
jogador -> gols
Hélio -> 20
Andrey -> 15
João -> 12
Antônio -> 11
```

Mas você deve estar se perguntando: como faço para descobrir a codificação de um arquivo de texto? Um forma simples de verificar é a seguinte: abra o arquivo com o NOTEPAD e depois escolha a opção **Salvar como...** Na janela que será aberta, observe o que está escrito na caixa **Codificação** (Figura 15). Se estiver escrito “UTF-8”, você precisará utilizar o parâmetro `encoding="utf-8"`; se estiver escrito “ANSI” você não precisará. Depois de verificar, basta clicar no botão **Cancelar**.

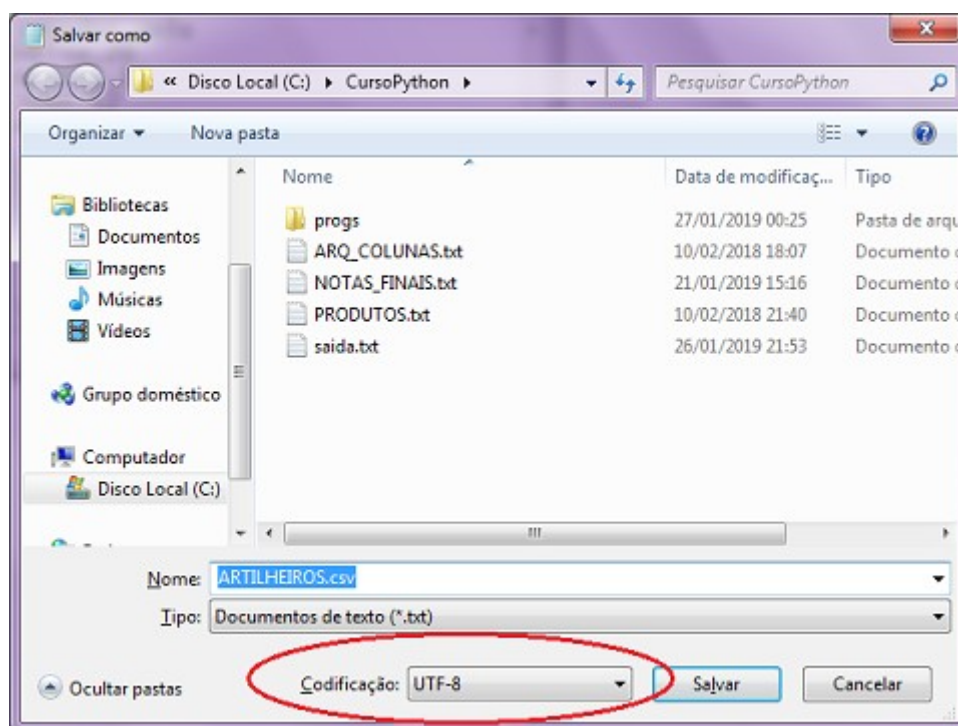


Figura 15. Verificando o encoding de um arquivo texto no Notepad



Lição 35 – Arquivos Texto: Processando Arquivos JSON

35.1 - Introdução

JSON (*JavaScript Object Notation*) é um modelo para **armazenamento** e **transmissão** de informações no formato texto. Apesar de muito simples, é o mais utilizado por aplicações Web devido a sua capacidade de estruturar informações de uma forma bem mais **compacta** do que a conseguida por seu maior rival, o modelo XML, tornando mais rápido o *parsing* (processamento e interpretação) dessas informações. Isto explica o fato de o JSON ter sido adotado por empresas como Google e Yahoo, cujas aplicações precisam transmitir grandes volumes de dados.

35.2 - Sintaxe

A ideia utilizada pelo JSON para representar informações é tremendamente simples: para cada valor representado, atribui-se um nome (ou rótulo) que descreve o seu significado. Esta sintaxe é derivada da forma utilizada pela linguagem JavaScript para representar informações. Por exemplo, para representar o ano de 2020, utiliza-se a seguinte sintaxe:

```
"ano": 2020
```

Você provavelmente percebeu que a sintaxe é muito parecida com a utilizada pela ED **dicionário**: temos uma chave mapeada para um valor. No JSON, um par **nome:valor** deve ser representado pelo nome entre aspas duplas, seguido de dois pontos, seguido do valor (do mesmo modo que fazemos ao especificar dicionários). Os valores podem possuir apenas **três tipos básicos**: numérico (inteiro ou real), booleano (`true` ou `false`, em minúsculo) e string. Existe ainda o valor `null`, para representar informação ausente. A seguir são apresentados alguns exemplos de informações representadas em JSON.

```
"nome": "Jane"
"sobrenome": "Austen"
"idade": 41
"nota": 9.9
"aprovado": true
```

A partir dos tipos básicos, é possível construir **tipos complexos**: **array** e **objeto**. Os arrays são delimitados por colchetes, com seus elementos separados entre vírgula (sintaxe idêntica a das listas do Python). Abaixo um exemplo de array de strings:

```
["RJ", "SP", "MG", "ES"]
```

Agora um exemplo de representação JSON para uma matriz de inteiros, neste caso a matriz identidade de ordem 3. Veja que a representação é idêntica a que usamos para listas bidimensionais.

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

Os objetos são especificados entre chaves e podem ser compostos por múltiplos pares nome:valor, por arrays e também por outros objetos. Desta forma, um objeto JSON pode representar,

virtualmente, qualquer tipo de informação! O exemplo abaixo mostra como podemos utilizar o JSON para representar dados de um filme.

```
{
  "titulo": "JSON x CSV",
  "resumo": "o duelo entre dois formatos para representar informações",
  "ano": 2020,
  "genero": ["aventura", "ação", "ficção"]
}
```

É possível representar mais de um objeto ou registro de uma só vez, bastando para isso utilizar um array. No exemplo abaixo, apresenta-se a forma para representar dois filmes em JSON:

```
[
  {
    "titulo": "JSON x CSV",
    "resumo": "o duelo entre dois formatos para representar informações",
    "ano": 2020,
    "genero": ["aventura", "ação", "ficção"]
  },
  {
    "titulo": "JSON James",
    "resumo": "a história de uma lenda do velho oeste",
    "ano": 2018,
    "genero": ["western"]
  }
]
```

Programa 52 – Importação de Arquivos JSON. No programa a seguir, utilizaremos o módulo ‘json’ (mais um módulo da *standard library*) para processar o arquivo “cinema.json”. Considere que o conteúdo do arquivo seja idêntico ao exemplo que acabamos de apresentar, contendo as informações dos filmes “JSON x CSV” e “JSON James”.

```
#P052: importando um arquivo JSON
import json

#(1)-Importa o arquivo JSON para a memória
nomeArq = 'C:/CursoPython/cinema.json'
with open(nomeArq) as f:
    filmes = json.load(f)

#(2)-Processa cada filme sequencialmente
print('-----')
print('Tipo da variável "filmes":', type(filmes))
print('Total de filmes = ', len(filmes))

for filme in filmes:
    print('-----')
    print('Tipo da variável "filme":', type(filme))
    print('Titulo:', filme['titulo'])
    print('Resumo:', filme['resumo'])
    print('Ano:', filme['ano'])
    print('Gênero(s):', filme['genero'])
```

Saída [52]:

Tipo da variável "filmes": <class 'list'>

Total de filmes = 2

Tipo da variável "filme": <class 'dict'>

Título: JSON x CSV

Resumo: o duelo entre dois formatos para representar informações

Ano: 2020

Gênero(s): ['aventura', 'ação', 'ficção']

Tipo da variável "filme": <class 'dict'>

Título: JSON James

Resumo: a história de uma lenda do velho oeste

Ano: 2018

Gênero(s): ['western']

O programa está dividido em duas partes:

- Na primeira parte temos a “receita de bolo” para a leitura de arquivos JSON: utiliza-se o método `load()` que, ao ser chamado, carrega o arquivo inteiro para a memória, estruturando-o em uma **lista de dicionários**. Veja que interessante: o arquivo “cinema.json” (arquivo contendo dois filmes) foi inteiramente importado para uma lista em que cada elemento (filme) é um dicionário. O dicionário possui 4 chaves: “título” (string), “resumo” (string), “ano” (int) e “generos” (lista). Isso demonstra a flexibilidade do formato JSON (e do Python) para representar informações complexas de uma maneira bastante natural.
- Na segunda parte, usamos o comando `for` para iterar sobre a nossa lista de filmes. A cada iteração, um filme é recuperado e suas propriedades podem ser exibidas utilizando a sintaxe normal para indexar dicionários (apresentada na Lição 25).
- Duas observações importantes: (i) por padrão, o módulo ‘json’ assume que o arquivo possui o *encoding* ‘ANSI’. Para abrir um arquivo no formato UTF-8, utilize o parâmetro `encoding="utf8"` dentro do comando `open()`; (ii) os arquivos JSON devem ser abertos com o uso do comando `with`.

Programa 53 – Gravação de Arquivos JSON. A seguir, apresenta-se um programa que grava o conteúdo de uma lista de dicionários para um arquivo JSON, através do método `dump()`.

```
#P053: gravando um arquivo JSON
```

```
import json
```

```
#(1)-Cria a lista de dicionários
```

```
filmes = []
```

```
filmes.append({  
    'titulo': 'Noel, Poeta da Vila',  
    'resumo': 'O filme conta a história do compositor Noel Rosa',  
    'ano': 2006,  
    'genero': ['Biografia', 'Musical']  
})
```

```

filmes.append({
    'titulo': 'Edukators',
    'resumo': 'De forma criativa, dois jovens lutam contra o sistema',
    'ano': 2004,
    'genero': ['Ação', 'Drama', 'Suspense']
})

filmes.append({
    'titulo': 'Um Conto Chinês',
    'resumo': 'Argentino mal-humorado resolve ajudar chinês desesperado',
    'ano': 2011,
    'genero': ['Comédia', 'Drama']
})

# (2) - Exporta os dados para um arquivo
nomeArq = 'C:/CursoPython/tres_filmes.json'

with open(nomeArq, 'w') as f_saida:
    json.dump(filmes, f_saida, ensure_ascii=False)

print('Arquivo gravado com sucesso...')

```

Saída [53]:
Arquivo gravado com sucesso...

35.3 – Comentários Finais

Nesta lição abordamos apenas o “feijão com arroz” sobre o assunto. Alguns tópicos importantes, porém um pouco mais “densos” sobre JSON foram deixados de lado (por exemplo, serialização e desserialização). Para encerrar, um resumo sobre as principais características do formato JSON:

- Utilizado para representar informações no formato texto;
- Possui natureza auto-descritiva (ou seja, basta “bater o olho” em um arquivo JSON para entender o seu significado);
- É capaz de representar de maneira simples e natural informações complexas. Alguns exemplos: objetos compostos (objetos dentro de objetos), relações de hierarquia, arrays, dados ausentes, etc.. Esse tipo de informação é muito difícil de ser representada no formato tabular;
- É um padrão *de facto* para representação de dados: foi formalizado na RFC 4627.
- É independente de linguagem de programação. Dados representados em JSON podem ser acessados por qualquer linguagem de programação, através de bibliotecas específicas (no caso do Python, usamos o módulo ‘json’ da *standard library*).



Lição 36 – Expressões Regulares

36.1 Introdução

Uma **expressão regular** (**regex**) pode ser definida como uma **regra** que especifica um determinado **padrão** a ser pesquisado em um texto. Não apenas o Python, mas quase todas as linguagens de programação modernas, além de grande parte dos editores de texto, oferecem essa ferramenta aos seus usuários.

Embora as expressões regulares sejam muito poderosas, elas são um pouquinho complicadas. Não seria exagero dizer que elas são quase uma linguagem de programação (não chega a tanto mais é quaaaaase). Sem dúvida, leva algum tempo até uma pessoa pegar o jeito e se tornar fluente na leitura e escrita de regexps! Mas vale muito a pena estudar o tema, pois trata-se de uma técnica muito empregada em processos de mineração de texto²¹, *web scraping*²², e outras tarefas de ciência de dados relacionadas à análise de documentos texto. Existem livros inteiros que tratam de regexps e, nesta lição, apresentaremos apenas um “saborzinho” sobre o tema, para que você possa ao menos “sair do zero”.

Antes de começarmos, uma informação: o módulo ‘re’ precisa ser importado em seu programa para que você possa trabalhar com expressões regulares. Ou seja, você terá que colocar o comando abaixo:

```
import re
```

36.2 Função `search()`

O módulo ‘re’ possui um bom número de funções²³, sendo `search()` a mais conhecida. Como o seu próprio nome indica, ela procura um padrão em um texto e retorna `True` se o mesmo for encontrado (`False`, caso contrário).

Programa 54 – Regex - Função `search()`. Considere o arquivo texto “dez_filmes.txt”, que, como o nome indica, armazena informações sobre dez diferentes filmes:

```
Corra, Lola, Corra (1998)|Ação, Aventura, Suspense
Edukators (2004)|Ação, Drama, Suspense
A Família Bélier (2014)|Comédia, Drama
O Filho da Noiva
Hair (1979)|Drama, Guerra, Musical
Jane Eyre (1943)|Drama, Romance
La La Land |Comédia, Musical, Romance
Nise: O Coração da Loucura (2015)|Biografia, História
As Pontes de Madison |Drama, Romance
Noel - Poeta da Vila |Biografia, Musical
```

21 Mineração de texto é um processo que utiliza algoritmos capazes de analisar coleções de documentos texto com objetivo de extrair padrões escondidos e potencialmente úteis.

22 *Web scraping* é uma técnica em que um programa que “finge” ser um navegador Web com o objetivo recuperar automaticamente o conteúdo páginas da Internet. O programa é comumente chamado de “robôzinho”.

23 <https://docs.python.org/3/library/re.html>

Veja que o arquivo é um pouco bagunçado. Ele não é bem um CSV, muito menos JSON, tampouco separado por colunas. Mas, de uma certa forma, possui algum padrão. Primeiro, observe que cada linha refere-se a um filme distinto. Alguns dos filmes possuem título, ano de produção entre parênteses, depois o delimitador "|" seguido de uma lista de gêneros. Outros possuem apenas o título, o separador "|" e os gêneros. Existe ainda um único filme sem a informação do ano e nem dos gêneros ("O Filho da Noiva").

No exemplo abaixo, apresenta-se um programa em que abre o arquivo "dez_filmes.txt" e joga todo o seu conteúdo para uma variável string chamada "conteudo". Depois, essa string é "quebrada" em uma lista (cada linha do arquivo vira um elemento da lista). Por fim, realiza-se um *loop* sobre a lista e utiliza-se `search()` para identificar e imprimir apenas as linhas que contêm a palavra 'Musical'.

```
#P054 Regexp basicon
import re

#(1)-Importa todo o arquivo para um stringão e depois o converte para uma lista
#      (cada linha do arquivo vira um elemento da lista)
nomeArq = 'C:/CursoPython/dez_filmes.txt'
f = open(nomeArq)
conteudo = f.read()
lst_filmes = conteudo.split("\n")

#(2)-Usa search() para encontrar as linhas que possuem a palavra "Musical"
for linha in lst_filmes:
    if re.search('Musical', linha):
        print(linha)
```

Saída [54]:

Hair (1979)|Drama, Guerra, Musical
La La Land |Comédia, Musical, Romance
Noel – Poeta da Vila |Biografia, Musical

36.3 Metacaracteres

O exemplo anterior não demonstra nem de longe verdadeiro poder das expressões regulares, pois poderíamos ter resolvido o mesmo problema utilizando simplesmente o método `find()`, disponível para variáveis do tipo string (ver Lição 26). Na realidade, quando trabalhamos com *regexps*, a coisa fica legal mesmo a partir do momento em que definimos regras que utilizam **metacaracteres**. Os metacaracteres são símbolos especiais que possuem funções específicas e podem ser combinados para gerar expressões complexas. É possível classificá-los²⁴ em quatro tipos distintos: âncoras, representantes, quantificadores e outros. Os metacaracteres pertencentes a cada categoria são apresentados, explicados e exemplificados nos quadros 17, 18, 19 e 20.

24 Classificação apresentada em “Expressões Regulares”, de Aurélio Jargas, autor dos melhores materiais que já vi sobre expressões regulares. Consulte: <https://aurelio.net/regex/guia/>.

Quadro 17. Metacaracteres do tipo âncora

Objetivo: servem para marcar posições específicas em uma linha

- ^ Marca o **início** de uma linha.
 - '^A' : obtém linhas que começam com a letra "A" (maiúscula).
- \$ Marca o **fim** de uma linha.
 - 'Drama\$' : obtém linhas que terminam com "Drama".

Quadro 18. Metacaracteres do tipo representante

Objetivo: utilizados para representar um ou mais caracteres

- . Representa um caractere **qualquer** (coringa)
 - '19.3' : casa com '1943', '1983', '1903', '19A3', '19Z3', etc.
 - 'B.la' : casa com 'Bela', 'Bala', 'Bola', etc.
 - 'Com.dia' : casa com 'Comédia', 'Comedia', 'Comxdia', 'Com1dia', etc.
- [...] Representa uma **lista** de caracteres **permitidos**.
 - '[Dd]rama' : casa apenas com 'Drama' e 'drama'.
 - '201[123]' : casa apenas com '2011', '2012', '2013'.
 - '<[Bb]>' : casa apenas com '' e ''.
- [^...] Representa uma **lista** de caracteres **não permitidos**.
 - '[^Dd]rama' : Nunca casa com 'Drama' e nem com 'drama' (mas poderia casar com 'Trama', por exemplo).

Quadro 19. Metacaracteres do tipo quantificador

Objetivo: servem para indicar o número de repetições permitidas para um determinado padrão, que deve ser especificado **à esquerda** do metacaractere.

- ? **Zero ou uma** repetição.
 - 'Eyr?e' : casa com 'Eye' e 'Eyre'.
 - '[Ii]phone[45]?' : casa com 'Iphone', 'iphone', 'Iphone4', 'iphone4', 'Iphone5' e 'iphone5'.
- * **Zero, uma ou mais** repetições.
 - 'Dra*ma' : casa com 'Drma', 'Drama', 'Draama', 'Draaama', etc.
 - '2*0' : casa com '0', '20', '220', '2220', etc.
- + **Uma ou mais** repetições.
 - 'Dra+ma' : casa com 'Drama', 'Draama', 'Draaama', etc.
 - '2+0' : casa com '20', '220', '2220', etc.
- {n,m} No mínimo **n** e no máximo **m** repetições.
 - 'dr[a]{2,3}ma' : casa apenas com 'draama' e 'draaama'.

Quadro 20. Outros metacaracteres

	Utilizado quando queremos especificar alternativas (ou). <ul style="list-style-type: none">'2004 2014' : casa com '2004' ou '2014'.'[Dd]rama [Rr]omance [Cc]omédia' : casa com 'Drama', 'drama', 'Romance', 'romance', 'Comédia' ou 'comédia'.
()	Utilizado para agrupar caracteres ou metacaracteres, permitindo a especificação de regexps mais complexas (grupo). <ul style="list-style-type: none">'A(n m)(e a)' : casa com 'Ana', 'Ane', 'Ama' e 'Ame'.
\	Chamado de "escape" , é utilizado para indicar que um determinado símbolo utilizado como metacaractere, como " ", ".", "^", "\$", etc., deverá ser tratado como um caractere normal (escape). <ul style="list-style-type: none">'\\ Comédia' : casa com ' Comédia'.
\s	Espaço em branco <ul style="list-style-type: none">'\sDrama' : casa apenas com ' Drama' (espaço em branco + 'Drama').
\S	Qualquer coisa que não seja um espaço em branco <ul style="list-style-type: none">'\SDrama' : Não casa jamais com ' Drama' (espaço em branco + 'Drama'), mas casa com " Drama", por exemplo.

Também é possível especificar intervalos de caracteres utilizando o sinal “-”, conforme apresentado no Quadro 21.

Quadro 21. Intervalos

[A-Z]	Lista com as letras sem acento (em maiúsculo);
[a-z]	Lista com as letras sem acento (em minúsculo);
[A-Za-z]	Lista com todas as letras não acentuadas, independente de estarem em maiúsculo ou minúsculo;
[0-9]	Digitos de 0 a 9.

O grande poder das expressões regulares está na possibilidade de combinar todas essas funcionalidades em uma única regra, como mostra o programa a seguir.

Programa 55 – Regexps com metacaraceres e intervalos. O programa abaixo apresenta diversos exemplos de expressões regulares que usam metacaracteres para efetuar buscas por padrões bem específicos na base de dados de filmes.

```
#P055 Regexps com metacaracteres
```

```
import re
```

```
#(1)-Importa todo o arquivo para um stringão e depois o converte para uma lista  
# (cada linha do arquivo vira um elemento da lista)
```

```
nomeArq = 'C:/CursoPython/dez_filmes.txt'
```

```
f = open(nomeArq)
```

```
conteudo = f.read()
```

```
lst_filmes = conteudo.split("\n")
```

```

#(2a)-Encontra os filmes que começam com "O" + espaço ou "A" + espaço
print('-----')
print('* * Filmes que começam com "O"+espaço ou "A"+espaço')

for linha in lst_filmes:
    if re.search('^(O|A)\s', linha):
        print(linha)

#(2b)-Encontra os filmes que possuem ano cadastrado
print('\n-----')
print('* * Filmes com ano cadastrado')

for linha in lst_filmes:
    if re.search('\([0-9]+\)', linha):
        print(linha)

#(2c)-Encontra os filmes que possuem alguma palavra com "a" e depois "r".
#(independente do número de letras entre "a" e "r")

print('\n-----')
print('* * Filmes com palavras que tenham "a" e depois "r"')

for linha in lst_filmes:
    if re.search('[A-Za-z]*a[A-Za-z]*r', linha):
        print(linha)

```

Saída [55]:

```

-----
* * Filmes que começam com "O"+espaço ou "A"+espaço
A Família Bélier (2014)|Comédia, Drama
O Filho da Noiva |Comédia, Drama

```

```

-----
* * Filmes com ano cadastrado
Corra, Lola, Corra (1998)|Ação, Aventura, Suspense
Edukators (2004)|Ação, Drama, Suspense
A Família Bélier (2014)|Comédia, Drama
Hair (1979)|Drama, Guerra, Musical
Jane Eyre (1943)|Drama, Romance
Nise: O Coração da Loucura (2015)|Biografia, História

```

```

-----
* * Filmes com palavras que tenham "a" e depois "r"
Edukators (2004)|Ação, Drama, Suspense
Hair (1979)|Drama, Guerra, Musical

```

Explicação:

- A expressão regular `^(O|A)\s` casa com qualquer linha que comece com a letra "O" ou a letra "A" e que, logo depois, possua um espaço em branco. Veja que há um filme que começa com "As" e que ficou de fora ("As Pontes de Madison"), pois queríamos apenas começando por "O"+espaço ou "A"+espaço.

- A expressão regular '`\([0-9]+\)`' casa com qualquer linha que possua um parêntese em aberto (veja que é preciso usar o escape “\”, pois o parêntese é um metacaractere), depois uma sequência com ou mais números e depois um parêntese fechado (mais uma vez “escapeado” com “\”). Com isso, a `regex` é capaz de recuperar os filmes que possuem o ano informado.
- Por fim, a complicada expressão regular '`[A-Za-z]*a[A-Za-z]*r`' casa com qualquer linha que inicie por zero ou mais letras maiúsculas ou minúsculas não acentuadas (`[A-Za-z]*`), depois tenha a letra "a" minúsculo, depois zero ou mais letras maiúsculas ou minúsculas não acentuadas (`[A-Za-z]*`) e depois tenha a letra "r" minúsculo!

36.4 Função `findall()`

A função `findall(r,s)` serve para extrair todas as substrings de uma string `s` que casem com a `regex` especificada em `r`. A função retorna uma lista contendo todas as substrings identificadas (caso nenhuma seja encontrada, retorna uma lista vazia). Ou seja: essa função não apenas serve para encontrar padrões em um texto, mas também para os extrair.

Programa 56 – Função `findall()`. Uma das abordagens mais utilizadas em aplicações de mineração de texto é a conhecida como **bag of words** (saco de palavras). Nesta abordagem, **cada palavra** presente em um *corpus* (conjunto de documentos) é transformada em uma **variável**. O código abaixo mostra como a função `findall()` pode ser utilizada para identificar e extrair todas as palavras presentes nos títulos de cada filme. Todas as pontuações são descartadas, mantendo-se apenas as letras.

```
#P056 Regex com findall()
import re

#(1)-Importa todo o arquivo para um stringão e depois o converte para uma lista
#      (cada linha do arquivo vira um elemento da lista)
nomeArq = 'C:/CursoPython/dez_filmes.txt'
f = open(nomeArq)
conteudo = f.read()
lst_filmes = conteudo.split("\n")

#(2)-Transforma cada título de filme em uma "saco de palavras"
print('-----')
print('* * palavras em cada título de filme:')

for linha in lst_filmes:
    titulo = linha.split("|")[0] #pega apenas o título, descartando os gêneros
    saco_palavras = re.findall('[A-Za-záéíóúâêôãõçÇ]+', titulo)
    print(saco_palavras)
```

Saída [56]:

```
-----
* * palavras em cada título de filme:
['Corra', 'Lola', 'Corra']
['Edukators']
['A', 'Família', 'Bélier']
['O', 'Filho', 'da', 'Noiva']
['Hair']
```

```
['Jane', 'Eyre']  
['La', 'La', 'Land']  
['Nise', 'O', 'Coração', 'da', 'Loucura']  
['As', 'Pontes', 'de', 'Madison']  
['Noel', 'Poeta', 'da', 'Vila']
```

No nosso programa, dentro do laço **for**, primeiro separamos o título do filme de seus gêneros utilizando o método **split()**. Depois, fazemos a variável “saco_palavras” receber o resultado da função **findall()** aplicada sobre o título e utilizando a regexp `'[A-Za-záéíóúâêôãõçç]+'`. Esta regexp especifica que queremos encontrar todas as sequências formadas por uma ou mais letras dentre as que foram especificadas na lista entre colchetes. Como resultado, para cada título, o programa gera uma lista com as sequências encontradas (que na verdade, formam uma palavra).

Bem, vamos terminar por aqui. Conforme mencionado no início da lição, as expressões regulares não constituem um assunto trivial e o nosso objetivo aqui era apenas realizar uma apresentação sobre o assunto. Para se tornar um especialista em regexps você precisará estudar algum livro inteiramente dedicado ao tema.

Capítulo V. Banco de Dados e Linguagem SQL

Apesar de a maior parte das informações digitais estar disponibilizada no formato texto (documentos HTML, CSV, JSON, Word, etc.), é muito importante levar em consideração que a “maior parte” não significa “tudo”! Há um grande volume de dados relevantes que não está estruturado em arquivos texto, mas sim em tabelas de bancos de dados relacionais, como Oracle, MySQL e Microsoft SQL Server. Este é o típico caso dos dados coletados e produzidos por sistemas corporativos – aplicativos desenvolvidos para necessidades essencialmente empresariais, como sistema de RH, sistema de cobranças, catálogo de produtos, sistema de fornecedores, controle de estoque, sistema acadêmico, sistema de vendas, etc.

Os dados armazenados em bancos de dados relacionais devem ser consultados e atualizados através de uma linguagem especialmente desenvolvida para este fim, denominada SQL. Esta é, basicamente, a única linguagem que os bancos relacionais entendem! Sendo assim, torna-se importante para todo cientista de dados aprender um pouco de SQL e sobre como acessar bancos de dados relacionais dentro de programas Python. No presente capítulo, realizamos uma breve introdução a estes assuntos. Você notará que o texto está centrado em uma única (mas belíssima) instrução SQL denominada SELECT. Ela pode ser considerada a instrução SQL mais importante para ciência de dados simplesmente porque a sua função é recuperar os dados armazenados nas tabelas. Trata-se de uma instrução muito poderosa, que permite a especificação de diferentes critérios de filtragem de linhas, é capaz de combinar eficientemente informações armazenadas em diferentes tabelas (operação conhecida como junção) e pode, até mesmo, produzir tabulações simples.

Nos exemplos apresentados ao longo do capítulo, o SQLite será o nosso principal instrumento de trabalho – uma pequena biblioteca que possui integração direta com o Python e consegue “emular” um sistema gerenciador de banco de dados relacional com grande eficácia.



Lição 37 – Banco de Dados

37.1 Banco de Dados

De maneira simplificada, podemos definir **banco de dados** (BD) como um **repositório central** de informações que podem ser **consultadas** e/ou **atualizadas** por **diversos usuários** simultaneamente, normalmente por intermédio de algum aplicativo ou sistema. A Figura 16 retrata a ideia recém-apresentada.

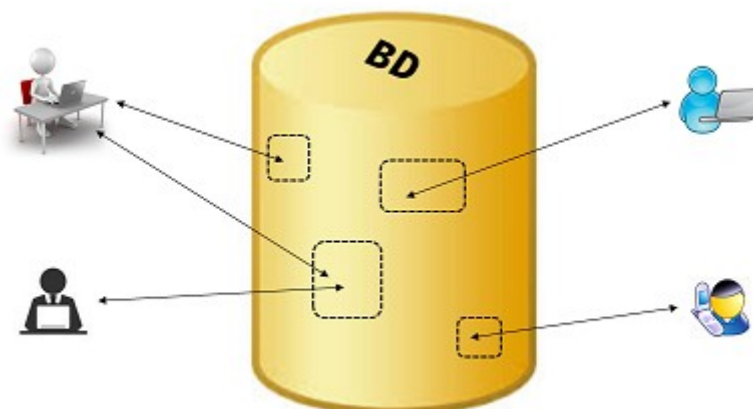


Figura 16. Um banco de dados e seus usuários

- No centro da Figura 16 encontra-se o BD propriamente dito: uma coleção de dados gerenciados por um computador denominado **servidor de BD**. Em geral, o servidor é uma máquina com hardware sofisticado e grande capacidade de armazenamento ou consiste em um *cluster* de computadores.
- O BD é quase sempre composto por uma coleção de dados sobre as operações e os negócios de uma empresa ou organização. Estes dados são produzidos pelos sistemas da empresa/organização, tais como: sistema de vendas, sistema de RH, sistema de atendimento via *call center*, sistema acadêmico (no caso de uma universidade), etc.
- A Figura 16 também mostra uma série de usuários on-line que interagem com o BD a partir de terminais remotos: computador da rede da empresa, computador doméstico com navegador da Internet, telefone celular, etc.
- Observe ainda que alguns usuários estão acessando simultaneamente as mesmas porções do banco (isto é, efetuando acessos concorrentes aos mesmos dados). Um exemplo prático deste tipo de situação ocorre, por exemplo, quando dois ou mais usuários de um sistema de vendas estão consultando o mesmo produto.

37.2 Sistemas Gerenciadores de Banco de Dados (SGBDs)

Um **Sistema Gerenciador de Banco de Dados (SGBD)** – *Database Management System* – pode ser definido como um software especial e muito sofisticado que permite a definição, construção, manipulação e compartilhamento de bancos de dados entre vários usuários e aplicações.

Ele também é utilizado para garantir a segurança dos dados, protegendo-os contra falhas (por exemplo, falhas de hardware) ou tentativas de acesso não autorizado. Embora existam diferentes modelos de SGBDs, o **modelo relacional** é o padrão dominante do mercado, sendo adotado pelos SGBDs mais utilizados, como Oracle, Microsoft SQL Server, PostgreSQL, MySQL e DB2.

Os SGBDs representam uma das categorias de software mais desenvolvidas e confiáveis dentre todas em que podemos classificar os sistemas de computação. A partir dos anos 80, o uso de SGBDs começou a se tornar prática comum nas empresas e hoje em dia é raro encontrar uma que não utilize um software deste tipo. Eles se tornaram tão populares que muitas vezes os termos “SGBD” e “banco de dados” são utilizados como sinônimos. No entanto, é importante esclarecer que os conceitos são diferentes – SGBD é na realidade o software que gerencia bancos de dados, ou seja, os bancos de dados residem nos SGBDs.

37.3 SQL

Os SGBDs relacionais são manipulados através de uma linguagem padrão desenvolvida especialmente para o ambiente relacional, denominada **SQL** (*Structured Query Language*). Muitos pesquisadores da área de banco de dados consideram-na a principal responsável pela imensa popularidade conquistada pelos SGBDs relacionais nos últimos 30 anos. A SQL é oferecida em praticamente todos os ambientes de programação e está disponível até mesmo em softwares de estatística como R e SAS. Esta linguagem é composta por um reduzido conjunto de instruções que permitem manipular um BD com diferentes finalidades. As principais instruções da SQL são apresentadas no Quadro 22.

Quadro 22. Principais instruções da SQL e sua diferentes finalidades

Instruções	Finalidade	Descrição
SELECT	Recuperação de Dados	Recupera registros armazenados em tabelas do banco de dados.
INSERT DELETE UPDATE	Manipulação de Dados	Inserção, alteração e remoção de registros de tabelas do banco de dados. Este subconjunto de instruções da SQL é conhecido como DML (<i>Data Manipulation Language</i>).
CREATE ALTER DROP	Definição de Dados	Criação, alteração e exclusão de objetos do banco de dados (ex.: tabelas, índices, etc.). Este subconjunto de instruções é conhecido como DDL (<i>Data Definition Language</i>).
COMMIT ROLLBACK	Controle de Transações	Gerenciam as modificações realizadas pelos comandos DML. Permitem agrupar as alterações dos dados em transações lógicas.
GRANT REVOKE	Controle de Acesso	Atribuem ou removem direitos de acesso ao banco de dados e seus objetos. São conhecidas como instruções DCL (<i>Data Control Language</i>).

Banco de dados é um assunto extremamente rico e abrangente. Tanto que no curso de graduação em Ciência da Computação, o estudo dos aspectos teóricos e práticos sobre banco de dados costuma ocupar duas ou mais disciplinas da grade, como, por exemplo, Banco de Dados I,

Banco de Dados II, Projeto de Banco de Dados, Bancos de Dados NoSQL, e por aí vai. Os livros²⁵ adotados pelas universidades normalmente passam das mil páginas! Por este motivo, optamos por uma abordagem bem mais objetiva no presente trabalho. Nos concentraremos em um único tema do universo dos BDs: a instrução **SELECT** da linguagem SQL. Esta instrução é utilizada para recuperar dados de tabelas, permitindo a especificação de filtros, a junção de tabelas e sendo até mesmo capaz de produzir agregações. **SELECT** é disparadamente a instrução mais utilizada em processos de análise de dados! Isso porque, no cenário mais comum, o cientista de dados não terá autorização para “invadir” o banco de dados de um sistema corporativo, criando novas tabelas e modificando os dados existentes. De maneira oposta, a ele será dado apenas o direito (GRANT) de selecionar o conjunto de dados de interesse para o estudo que deseja realizar. E o único comando da SQL destinado à seleção de dados é exatamente nosso querido **SELECT**. Fim de papo!

Nas próximas lições, mostraremos como utilizar o comando **SELECT** sobre pequenas bases de dados criadas no SQLite que fazem parte do material de apoio de nosso livro. O SQLite foi escolhido como nosso SGBD de trabalho porque possui uma série de características interessantes, como: (i) já está automaticamente disponível para uso no Python, não exigindo a instalação e configuração de nenhuma biblioteca; (ii) suporta a linguagem SQL em quase sua totalidade; (iii) gera os bancos de dados em arquivos binários únicos e portáteis, ou seja, você pode criar um BD em um computador Windows e depois utilizá-lo em uma máquina com Linux, um iPhone, um celular com Android, etc. (um único arquivo SQLite pode armazenar até 2 terabytes de dados!).

A biblioteca 'sqlite3', da *standard library*, permite com que programas Python possam manipular BDs criados no SQLite através da linguagem SQL. Para utilizá-la em seus programas, realize a importação da seguinte maneira:

```
import sqlite3
```



Lição 38 – SQL: SELECT Básico

38.1 Apresentando a Base de Dados “RH.db”

Nesta lição, serão apresentados alguns exemplos de utilização básica do comando **SELECT**. Trabalharemos com um pequeno BD SQLite formado por duas tabelas, chamadas *Profissao* e *Funcionario*, apresentadas na Figura 17.

- A tabela *Profissao* contém uma relação de “ids” (códigos) e “nomes” de profissões, enquanto *Funcionario* armazena a “matrícula”, “nome”, “idade”, “sexo” e o “id da profissão” de diferentes funcionários de uma empresa. Sendo assim, *Profissao* possui duas colunas, enquanto *Funcionario* possui cinco. No jargão da área de BD, as colunas de tabelas são chamadas de **atributos**.
- A coluna “id” de *Profissao* é utilizada para identificar unicamente cada **registro** (linha) da tabela. Ela é chamada de **chave primária** da tabela. De maneira análoga, a coluna “mat” é a chave primária de *Funcionario*.

25 Um exemplo: Elmasri, R; Navathe S. B. Fundamentals of Database Systems, 7th ed., Pearson, 2016.

- As colunas “id_prof” de *Funcionario* e “id” de *Profissao*, servem como atributo de ligação entre ambas as tabelas. Mais especificamente, o campo “id_prof” de *Funcionario* é uma **chave estrangeira** referenciando o campo “id” de *Profissao*. Isto significa que existe uma restrição de integridade que estabelece que a profissão de um funcionário deve respeitar algum dos códigos cadastrados em *Profissao* ou possuir valor NULL (o NULL da SQL é basicamente a mesma coisa que o None do Python).

<i>id</i>	<i>nome</i>
1	Engenheiro
2	Desenvolvedor
3	Cientista de Dados
4	Minerador de Dados
5	Matemático

<i>mat</i>	<i>nome</i>	<i>idade</i>	<i>sexo</i>	<i>id_prof</i>
M01	George	58	M	5
M02	Jane	32	F	3
M03	Aldous	40	M	3
M04	Thomas	28	M	1
M05	Mary	43	F	NULL

Figura 17. Tabelas “Profissao” e “Funcionario”

As duas tabelas foram criadas em uma base de dados SQLite chamada “RH.db”, que faz parte do material de apoio de nosso curso. Caso você ainda não tenha baixado o material, faça-o por favor, pois ele será necessário para a execução dos exemplos contidos no capítulo.

Programa 57 – Conexão do Python com o SQLite. Este exemplo mostra a receita básica para conectar o Python com um BD SQLite e consultá-lo via SQL. Utilizamos como exemplo o comando SQL para retornar todas as linhas e colunas da tabela *Funcionario*:

```
SELECT * FROM Funcionario
```

Conforme indicado acima, para exibir todas as colunas de uma tabela, deve-se utilizar a palavra-chave **SELECT** com um “*” (asterisco). A tabela a ser consultada deve ser indicada após a palavra chave **FROM**.

```
#P057 SELECT *
import sqlite3

# (1) - Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)
```

```

# (2)-Executa o SQL
c = minha_conn.cursor()
c.execute('SELECT * FROM Funcionario')

# (3)-Exibe os resultados

# (3.1)-Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

# (3.2)-Obtém e exibe cada linha recuperada do BD
for linha in c:
    print(linha)

# (4)-Fecha a conexão
minha_conn.close()

```

Saída [57]:

```

('mat', 'nome', 'idade', 'sexo', 'id_prof')
('M01', 'George', 58, 'M', 5)
('M02', 'Jane', 32, 'F', 3)
('M03', 'Aldous', 40, 'M', 3)
('M04', 'Thomas', 28, 'M', 1)
('M05', 'Mary', 43, 'F', None)

```

A seguir, uma explicação detalhada de cada linha do programa:

- **import** sqlite3
 - Importa o módulo “sqlite3”.
- `minha_conn = sqlite3.connect('C:/CursoPython/RH.db')`
 - Utiliza a função **connect()** do módulo ‘sqlite3’ para criar uma “**conexão**” para o banco de dados SQLite estruturado no arquivo “RH.db”.
 - O nome “conexão” é usado porque na maioria das vezes – especialmente se estivermos usando SGBDs “de verdade” como Oracle ou SQL Server – o BD estará armazenado em um servidor de banco de dados, que corresponde a uma máquina diferente daquela onde o programa Python está sendo executado. No entanto, quando trabalhamos com o SQLite, o BD normalmente consistirá em um arquivo local na mesma máquina.
- `c = minha_conn.cursor()`
 - Utiliza a função **cursor()** do módulo ‘sqlite3’ para criar um **cursor** associado à conexão. Um *cursor* é algo bem parecido com um *file handle*, mas que irá nos permitir executar operações sobre dados de um BD em vez de um arquivo. Sendo assim, abrir um cursor é conceitualmente similar a abrir um arquivo texto com o comando **open()**.
- `c.execute('SELECT * FROM Funcionario')`
 - Uma vez aberto o *cursor*, podemos executar qualquer instrução SQL sobre o BD através da função **execute()**. Neste exemplo, foi utilizada a instrução **SELECT ***, que retornará um conjunto de resultados composto por todas as linhas e colunas da tabela *Funcionario*. Veja que a instrução SQL é passada em uma string, como parâmetro para a função **execute()**.

- `nomes_colunas = next(zip(*c.description))`
- `print(nomes_colunas)`
 - Aqui estamos utilizando `description`, uma **propriedade**²⁶ dos objetos do tipo *cursor*, para pegar os nomes das colunas recuperadas pelo comando SQL. Utilizamos um “macete”, que envolve o uso da função `zip()` para jogar os nomes das colunas em uma estrutura de dados do tipo tupla. Não daremos uma explicação detalhada sobre a função `zip()`, que não faz parte do escopo de nosso livro. Basta apenas você utilizar a receita acima e sempre conseguirá obter os nomes das colunas de um resultado obtido através da execução da instrução `SELECT`.
- `for linha in c:`
 - Varre todo o conjunto de resultados de forma sequencial, da primeira à última linha. Muito simples e intuitivo, não? É importante comentar que cada linha retornada da tabela, é guardada em uma **tupla** Python. Isto faz todo sentido, uma vez que as tuplas são imutáveis e, quando trabalhamos com o `SELECT`, não desejamos alterar os dados originais do BD, mas apenas obter um conjunto de resultados para analisar no programa Python.
 - Assim como ocorre quando estamos varrendo um arquivo sequencialmente, o *cursor* só sabe “andar pra frente”. Uma vez que a gente passe por uma linha no comando `for`, não tem mais como voltar para a anterior.
- `c = minha_conn.close()`
 - Fecha a conexão com o banco de dados.

Programa 58 – Seleção de Colunas Específicas de uma Tabela

Para selecionar colunas específicas de uma tabela, os nomes das colunas em questão devem ser separados por vírgulas, logo após a palavra-chave `SELECT`. O exemplo a seguir mostra o comando para recuperar apenas o nome e idade dos funcionários.

```
#P058 Seleção de Colunas Específicas de uma Tabela
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Executa o SQL
c = minha_conn.cursor()
c.execute('SELECT nome, idade FROM Funcionario')

# (3)-Exibe os resultados

# (3.1)-Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

# (3.2)-Obtém e exibe cada linha recuperada do BD
```

²⁶ “Propriedade” é um conceito que pertence ao mundo da programação orientada a objetos. Consiste basicamente em uma variável utilizada para descrever a característica de um objeto.

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

# (4)-Fecha a conexão
minha_conn.close()
```

Saída [58]:
('nome', 'idade')
('George', 58)
('Jane', 32)
('Aldous', 40)
('Thomas', 28)
('Mary', 43)

Veja que o programa é quase idêntico ao anterior, possuindo uma única mudança: trocamos a string SQL a ser executada pelo cursor de 'SELECT * FROM Funcionario' para 'SELECT nome, idade FROM Funcionario'. Nos programas restantes desta lição, ocorrerá sempre a mesma coisa, nós precisaremos mexer apenas na instrução SELECT. Já temos a nossa receitinha pronta para buscar qualquer tipo de informação no BD!

Programa 59 – Restringindo o Conjunto de Linhas Retornadas (WHERE). Neste exemplo, utilizaremos uma instrução SELECT para recuperar o nome e idade dos funcionários cujo “id da profissão” é igual a 3 (ou seja, os funcionários cuja profissão é “Cientista de Dados”). Para tal, é preciso utilizar a cláusula **WHERE** com a condição `id_prof = 3`.

```
#P059 Restringindo o conjunto de linhas retornadas (WHERE)
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Executa o SQL
c = minha_conn.cursor()
c.execute('SELECT nome, idade FROM Funcionario WHERE id_prof=3')

# (3)-Exibe os resultados

# (3.1)-Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

# (3.2)-Obtém e exibe cada linha recuperada do BD
for linha in c:
    print(linha)

# (4)-Fecha a conexão
minha_conn.close()
```

Saída [59]:
('nome', 'idade')
('Jane', 32)
('Aldous', 40)

A instrução SQL utilizada neste exemplo solicita com que sejam recuperados o valores das colunas “nome” e “idade” da tabela *Funcionario* considerando apenas as linhas onde a coluna “id_prof” possui valor igual a 3. Como só há duas linhas que atendem a esta condição, apenas dois resultados são retornados no conjunto resposta.

É importante observar que a cláusula `WHERE` suporta a definição de condições complexas, concatenadas por operadores lógicos (`AND`, `OR`, `NOT`). Ou seja, a forma como uma condição `WHERE` é montada equivale a mesma forma utilizada no comando `if`.

Programa 60 – Ordenando os Resultados (ORDER BY). É possível especificar que os resultados sejam ordenados por uma ou mais colunas através do uso da cláusula `ORDER BY`. O exemplo seguinte mostra uma consulta que retorna os funcionários ordenados por idade.

```
#P060 SELECT com ORDER BY
import sqlite3

# (1) - Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

# (2) - Executa o SQL
c = minha_conn.cursor()
c.execute('SELECT * FROM Funcionario ORDER BY idade')

# (3) - Exibe os resultados

# (3.1) - Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

# (3.2) - Obtém e exibe cada linha recuperada do BD
for linha in c:
    print(linha)

# (4) - Fecha a conexão
minha_conn.close()
```

Saída [60]:

```
('mat', 'nome', 'idade', 'sexo', 'id_prof')
('M04', 'Thomas', 28, 'M', 1)
('M02', 'Jane', 32, 'F', 3)
('M03', 'Aldous', 40, 'M', 3)
('M05', 'Mary', 43, 'F', None)
('M01', 'George', 58, 'M', 5)
```



Lição 39 – SQL: Junção de Tabelas

As junções são utilizadas em instruções `SELECT` para recuperar dados de duas tabelas e “juntá-los” com o objetivo de produzir uma saída combinada. As tabelas envolvidas em uma junção precisarão possuir um atributo em comum. Em nosso exemplo, o “id da profissão” é quem “liga” as

tabelas *Profissao* e *Funcionario*, ou seja, é o tal atributo em comum. Veja que a SQL não exige que o atributo de ligação possua o mesmo nome nas duas tabelas que serão combinadas.

Programa 61 – INNER JOIN. A **junção interna** (INNER JOIN) seleciona todas as linhas de ambas as tabelas contanto que haja coincidência nos valores das colunas especificadas na **condição de junção**. No exemplo abaixo, o INNER JOIN é utilizado para obter a lista de todos os funcionários e os nomes de suas profissões.

```
#P061 SELECT com INNER JOIN
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Monta e Executa o SQL
c = minha_conn.cursor()

vSQL = """SELECT F.*, P.*
FROM Funcionario F INNER JOIN Profissao P
ON (F.id_prof = P.id)
"""

c.execute(vSQL)

# (3)-Exibe os resultados

# (3.1)-Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

# (3.2)-Obtém e exibe cada linha recuperada do BD
for linha in c:
    print(linha)

# (4)-Fecha a conexão
minha_conn.close()
```

Saída [61]:
('mat', 'nome', 'idade', 'sexo', 'id_prof', 'id', 'nome')
('M01', 'George', 58, 'M', 5, 5, 'Matemático')
('M02', 'Jane', 32, 'F', 3, 3, 'Cientista de Dados')
('M03', 'Aldous', 40, 'M', 3, 3, 'Cientista de Dados')
('M04', 'Thomas', 28, 'M', 1, 1, 'Engenheiro')

Antes de apresentarmos a explicação sobre o INNER JOIN propriamente dito, vamos falar de uma pequena novidade introduzida nesse programa: desta vez a instrução SELECT não foi especificada diretamente em `c.execute()`, mas sim em uma variável string chamada `vSQL`:

```
vSQL = """SELECT F.*, P.*
FROM Funcionario F INNER JOIN Profissao P
ON (F.id_prof = P.id)
"""
```


Adotamos esse procedimento simplesmente pelo fato de este `SELECT` ser mais comprido do que os anteriores. Veja que o definimos em 3 linhas! Aliás, para poder fazer a string ocupar mais de uma linha, foi necessário abri-la e fechá-la utilizando três aspas: `"""`. O Python trabalha dessa forma.

Agora sim, vamos falar do `INNER JOIN`, pois este é o assunto realmente importante. Para realizar a junção interna entre duas tabelas em SQL, basta seguir sempre a mesma receita:

1. Especificar os nomes das tabelas envolvidas na junção na cláusula `FROM`. Cada tabela poderá receber um **apelido**, utilizado para simplificar a referência às mesmas. Em nosso exemplo, *Funcionario* recebeu o apelido “F” e *Profissao* o apelido “P”
2. Especificar o tipo de junção `INNER JOIN` entre as duas tabelas.
3. Para efetivarmos a junção entre as duas tabelas, é preciso escrever a palavra-chave `ON` e depois especificar a **condição de junção** entre as duas tabelas (utilizamos parênteses para melhorar a legibilidade). Isso consiste em indicar qual é o atributo que “liga” as duas tabelas, ou seja, qual é a coluna que está presente nas duas tabelas. . Nesse caso, é o “id da profissão”, que se chama “id” em *Profissao* e que se chama “id_prof” em *Funcionario*.

A junção final ficou então definida como:

```
FROM Funcionario F INNER JOIN Profissao P ON (F.id_prof = P.id)
```

Observe que ao lado da palavra `SELECT`, os campos selecionados são agora precedidos pelo apelido de cada tabela, para indicar a que tabela pertence cada coluna (tabela “F” ou tabela “P”). Observe ainda que a funcionária “M05-Mary” não aparece no resultado final, pois ela não possui um código de profissão associado (está com o valor `NULL` armazenado na coluna “id_prof”).

Programa 62 – LEFT JOIN. A operação de `LEFT JOIN` (ou `LEFT OUTER JOIN`) retorna todas as linhas da tabela à esquerda, mesmo que não exista casamento (valor equivalente) na tabela à direita. Em nosso exemplo, a funcionária “M05-Mary” não possui uma profissão cadastrada. Se desejarmos produzir uma listagem contendo todos os funcionários com seus respectivos nomes de profissão, incluindo os funcionários sem profissão cadastrada, é preciso fazer uso do `LEFT JOIN`:

```
#P062 SELECT com LEFT JOIN
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Monta e Executa o SQL
c = minha_conn.cursor()

vSQL = """SELECT F.*, P.*
FROM Funcionario F LEFT JOIN Profissao P
ON (F.id_prof = P.id)
"""

c.execute(vSQL)

# (3)-Exibe os resultados
```

```
#(3.1)-Obtém e exibe os nomes das colunas
nomes_colunas = next(zip(*c.description))
print(nomes_colunas)

#(3.2)-Obtém e exibe cada linha recuperada do BD
for linha in c:
    print(linha)

#(4)-Fecha a conexão
minha_conn.close()
```

Saída [62]:

```
('mat', 'nome', 'idade', 'sexo', 'id_prof', 'id', 'nome')
('M01', 'George', 58, 'M', 5, 5, 'Matemático')
('M02', 'Jane', 32, 'F', 3, 3, 'Cientista de Dados')
('M03', 'Aldous', 40, 'M', 3, 3, 'Cientista de Dados')
('M04', 'Thomas', 28, 'M', 1, 1, 'Engenheiro')
('M05', 'Mary', 43, 'F', None, None, None)
```

Para implementar uma consulta com `LEFT JOIN` a receita é a mesma utilizada para o `INNER JOIN`. O único detalhe a ser mudado é que, na cláusula `FROM`, a tabela a qual desejamos levar todas as linhas para o resultado final deve ser especificada à esquerda. Em nosso exemplo, desejávamos todos os funcionários no resultado final, independentemente de terem profissão ou não, por isso a tabela *Funcionario* foi especificada à esquerda da palavra `LEFT JOIN`, enquanto *Profissao* ficou à direita. Já com relação a condição de junção (`ON`) a ordem não faz diferença, ou seja, tanto faz especificar “`ON (F.id_prof = P.id)`” ou “`ON (P.id = F.id_prof)`”.

Sendo assim, para listar todos os cargos e funcionários, mesmo os cargos que não tenham um funcionário associado, devermos especificar *Profissao* à esquerda.

```
SELECT F.*, P.*
FROM Profissao P LEFT JOIN Funcionarios F
ON (F.id_prof = P.id)
ORDER BY mat;
```

RIGHT JOIN e FULL JOIN

- A SQL também possui os tipos de junção `RIGHT JOIN` e `FULL JOIN`. Porém, nenhum deles é suportado pelo SQLite. A junção do tipo `RIGHT JOIN` retorna todas as linhas da tabela à direita, mesmo que não exista casamento (valor equivalente) na tabela à esquerda. Ou seja: o `LEFT JOIN` e o `RIGHT JOIN` funcionam da mesma maneira, a única diferença é que no primeiro caso iremos especificar a tabela a qual desejamos todas as linhas no resultado à esquerda e no segundo caso ela ficará à direita. Na prática o `LEFT JOIN` acabou se tornando mais adotado pelos programadores que trabalham com SQL.
- A junção do tipo `FULL JOIN` retorna todas as linhas da tabela à esquerda e todas as linhas da tabela à direita devidamente combinadas, de acordo com a especificação da condição de junção. Caso existam linhas na tabela esquerda que não tenham equivalência com a tabela direita, elas serão levadas para o resultado final. E caso existam linhas na tabela direita que não tenham equivalência na tabela esquerda, elas também serão levadas para o resultado final. Mostraremos um exemplo de operação de `FULL JOIN` no Capítulo VII, que cobre a biblioteca ‘pandas’.



Lição 40 – SQL: Produzindo Resultados Agregados

A instrução `SELECT` suporta a consultas complexas não somente com o uso de junções, mas também com funções para a produção de resultados agregados. Considere agregação como um processo de transformação de dados que produz valores escalares a partir de tabelas, geralmente com o uso de uma função matemática ou estatística (soma, média, etc.)

As funções para produção de resultados agregados serão apresentadas a partir de exemplos que exploram a tabela *Projeto*, armazenada no BD SQLite chamado “Reformas.db”, cuja estrutura e conteúdo são apresentados na Figura 18.

<i>codigo</i>	<i>tipo</i>	<i>local</i>	<i>custo</i>
P1	A	RJ	500.000
P2	A	DF	900.000
P3	B	RJ	150.000
P4	A	DF	1.000.000
P5	B	RJ	NULL
P6	A	SP	850.000

Figura 18. Tabela “Projeto”

Considere que esta tabela armazena os projetos correntemente conduzidos por uma empresa de reforma de edifícios. Ela possui quatro colunas: “codigo” (código do projeto), “tipo” (‘A’=Reforma Estrutural ou ‘B’=Manutenção de Fachada), “local” (UF onde obra está sendo realizada) e “custo” (orçamento da obra em R\$). Observe que o projeto ‘P5’ ainda não tem o orçamento definido (está com valor NULL armazenado na variável “custo”).

Programa 63 – COUNT. A função `COUNT(v)` da linguagem SQL determina o número de ocorrências de valores não-nulos de uma dada coluna em uma tabela. Por outro lado, `COUNT(*)` é utilizado para contar o número total de linhas de uma tabela:

```
#P063 SELECT com COUNT
import sqlite3

#(1)-Conecta com o BD
nomeBD = 'C:/CursoPython/Reformas.db'
minha_conn = sqlite3.connect(nomeBD)

#(2)-Executa o SQL e exibe os resultados
c = minha_conn.cursor()
vSQL = "SELECT COUNT(codigo), COUNT(custo), COUNT(*) FROM Projeto"
c.execute(vSQL)

for linha in c:
    print(linha)

#(3)-Fecha a conexão
minha_conn.close()
```

Saída [63]:

(6, 5, 6)

O primeiro valor 6 indica que este é o número de registros onde a variável “codigo” não possui valor NULL. Analogamente, o valor 5 indica que existem cinco registros em que a variável “custo” não está nula. Por fim, o último 6 vem de COUNT(*), indicando o número de linhas da tabela *Projeto*. Veja que neste exemplo não capturamos e exibimos os nomes das colunas do conjunto de resultados (mostramos apenas os resultados).

Programa 64 – MAX, MIN, AVG, SUM. Para uma determinada coluna de uma tabela, as funções MIN, MAX e AVG retornam, respectivamente, o valor mínimo, máximo e médio da mesma. Já a função SUM retorna a somatório dos valores da coluna em todas as linhas. Os valores NULL são ignorados por essas funções.

```
#P064 SELECT com MAX, MIN, AVG, SUM
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/Reformas.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Executa o SQL e exibe os resultados
c = minha_conn.cursor()
vSQL = "SELECT MIN(custo), MAX(custo), AVG(custo), SUM(custo) FROM Projeto"
c.execute(vSQL)
for linha in c:
    print(linha)

# (3)-Fecha a conexão
minha_conn.close()
```

Saída [64]:

(150000, 1000000, 680000.0, 3400000)

Programa 65 – GROUP BY. A cláusula **GROUP BY** pode ser combinada com as funções de grupo para permitir a produção de resultados agregados por uma ou mais colunas. O exemplo abaixo obtém os valores mínimo, máximo, médio e o somatório do custo por tipo de projeto.

```
#P065 SELECT com GROUP BY
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/Reformas.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Executa o SQL e exibe os resultados
c = minha_conn.cursor()
vSQL = """
SELECT
    tipo, MIN(custo), MAX(custo), AVG(custo), SUM(custo)
FROM Projeto
GROUP BY tipo
"""
```

```

c.execute(vSQL)
for linha in c:
    print(linha)

# (3)-Fecha a conexão
minha_conn.close()

```

Saída [65]:

```

('A', 500000, 1000000, 812500.0, 3250000)
('B', 150000, 150000, 150000.0, 150000)

```

Programa 66 – HAVING. Caso você queira eliminar algum grupo do resultado final produzido por um SELECT com GROUP BY, torna-se necessário utilizar a cláusula **HAVING**. No exemplo abaixo, são eliminados todos os grupos em que o somatório do custo é inferior à 600.000.

```

#P066 SELECT com GROUP BY e HAVING
import sqlite3

# (1)-Conecta com o BD
nomeBD = 'C:/CursoPython/Reformas.db'
minha_conn = sqlite3.connect(nomeBD)

# (2)-Executa o SQL e exibe os resultados
c = minha_conn.cursor()
vSQL = """
    SELECT
        tipo,
        local,
        SUM(custo)
    FROM Projeto
    GROUP BY tipo, local
    HAVING SUM(custo) >= 600000
    """

c.execute(vSQL)
for linha in c:
    print(linha)

# (3)-Fecha a conexão
minha_conn.close()

```

Saída [66]:

```

('A', 'DF', 1900000)
('A', 'SP', 850000)

```

Assim chegamos ao final do capítulo. Realizamos uma breve introdução à linguagem SQL focando nos usos mais comuns da instrução SELECT em programas de ciência de dados. Assim como ocorre com as expressões regulares, a linguagem SQL é um tema que, por si só, exige um livro inteiro para que possa ser apresentada em detalhes.

Capítulo VI. Biblioteca ‘NumPy’

A biblioteca ‘NumPy’ (*Numerical Python*) estende a linguagem Python com a estrutura de dados **ndarray** (*n-dimensional array*), direcionada para a computação de alto desempenho sobre vetores e matrizes. Ela oferece uma série de benefícios sobre a tradicional lista do Python (abordada no Capítulo III), dentre os quais se destacam a possibilidade de representar e manipular vetores e matrizes de uma forma muito mais natural, o ganho de desempenho na execução de funções matemáticas e estatísticas sobre grandes volumes de dados e a maior facilidade para ler e escrever dados em disco.

A ‘NumPy’ é considerada a “pedra fundamental” da computação científica em Python, não apenas pelas funcionalidades que oferece, mas principalmente pelo fato de as suas propriedades e métodos terem sido utilizados como base para o desenvolvimento de diversas outras bibliotecas importantes para ciência de dados, como a ‘SciPy’ (*Scientific Python*), ‘pandas’ (*Python Data Analysis Library*) e ‘Matplotlib’ (*Plotting Library*).

Este capítulo é inteiramente dedicado à ‘NumPy’. Apesar de, isoladamente, esta biblioteca fornecer poucas funções de alto nível para análise de dados (o “lance” dela é processar vetores e matrizes), há um consenso entre os *pythonistas* do mundo inteiro a respeito de sua grande importância. Isto porque, ao dominar a computação orientada por arrays da ‘NumPy’, você poderá utilizar outras bibliotecas para ciência de dados com muito mais eficácia. A relação a seguir apresenta os principais tópicos cobertos neste capítulo:

- Propriedades dos ‘ndarrays’;
- Diferentes formas para criar e popular vetores e matrizes;
- Operações de indexação, fatiamento e varredura de ndarrays.
- Aplicação de funções matemáticas e estatísticas sobre ndarrays;



Lição 41 – NumPy: Introdução

Um array é uma espécie de *container* que armazena um conjunto de valores de um mesmo tipo. Cada valor é armazenado em uma posição ou **célula** específica. A Figura 19 apresenta dois exemplos. Na parte de cima, temos um **array unidimensional**, ou **vetor** que contém uma coleção de números reais estruturados em 5 células (células 0 a 4). Embaixo dele, um **array bidimensional**, ou **matriz**, que armazena uma coleção de valores inteiros, estruturados em 12 células dispostas em 3 linhas e 4 colunas.

	[0]	[1]	[2]	[3]	[4]	
	9.1	7.2	5.5	10.0	6.9	array unidimensional

	[0]	[1]	[2]	[3]	
[0]	1	0	1	0	array bidimensional
[1]	0	1	0	1	
[2]	0	0	0	1	

Figura 19. um array unidimensional (vetor) e outro bidimensional (matriz)

É importante observar que a ‘NumPy’ também permite a definição de arrays com 3 ou mais dimensões, mas estes tipos de estruturas não serão abordadas em nosso livro. Seja qual for o número de dimensões, todo array ‘NumPy’ sempre estará associado às seguintes **propriedades** elementares:

- **dtype**: descreve o tipo de dado dos elementos contidos no array. Existem diferentes tipos, muito específicos e bem detalhados para representar inteiros, reais, números complexos, strings e booleans, conforme será detalhado na Lição 39.
- **ndim**: número de dimensões do array (1 para vetor, 2 para matriz, 3 se array é tridimensional, etc.).
- **shape**: indica o formato do array (basicamente, nos vetores indica o número de células, e nas matrizes indica o número de linhas e de colunas).
- **axes**: são os números dos eixos do array. Um vetor possui apenas a axis (eixo) 0. Por sua vez, para uma matriz, a ‘NumPy’ usa axis 0 associada à linha e axis 1 associada à coluna. Essa propriedade é muito importante em situações onde desejamos modificar o formato ou o número de dimensões de um array.
- **data**: indica o endereço de memória do primeiro byte do array.
- **strides**: número de bytes que deve ser “pulado” na memória para ir ao próximo elemento da matriz. Por exemplo, se os “strides” de uma matriz são (16,1), é preciso avançar 1 byte para alcançar a próxima coluna e 16 bytes para localizar a próxima linha. Ambas as propriedades,

“strides” e “data”, são utilizadas em situações específicas, onde o programador precisa manipular a memória de uma forma mais direta para otimizar o desempenho de um *script*.

A ‘NumPy’ não faz parte do Python padrão, isto é, ela não pertence à *standard library*. No entanto, está incluída em todas distribuições do Python voltadas para computação científica²⁷, como a WinPython e a Anaconda. Antes de utilizar a biblioteca ‘NumPy’ em um programa, você precisa usar o comando **import** da forma mostrada a seguir:

```
import numpy as np.
```

Veja que, além de importar a biblioteca, a renomeamos para “np”. Essa troca de nome não é algo obrigatório, mas acabou tornando-se uma convenção entre os *pythonistas*. Sendo assim, vamos seguir a convenção e sempre empregar o apelido “np” nos programas apresentados neste capítulo.



Lição 42 – NumPy: Criação de Vetores

Programa 67 – Criando um Vetor com o Método `np.array()`. Nesta lição, veremos algumas das diferentes técnicas que podem ser empregadas para a criação de vetores. A mais simples consiste em utilizar o método `np.array()` passando uma sequência como parâmetro. O exemplo a seguir cria um vetor de 10 posições, denominado “vet_notas”, a partir dos dados de uma lista. Considere que os dados em questão indiquem as notas médias de 10 alunos de um curso de Python.

```
#P067: Olá Vetor NumPy!
import numpy as np  #importa a biblioteca, renomeando-a para np
                    #(é uma convenção... todo mundo faz isso!)

#Neste exemplo, começamos criando uma lista (lst_notas)...
lst_notas = [7.8, 8.5, 10.0, 9.2, 5.0, 8.5, 6.4, 8.6, 7.5, 9.0]

#... e agora criamos um array a partir dessa lista (vet_notas)
vet_notas = np.array(lst_notas)

print('vet_notas = ', vet_notas)                #imprime o vetor
print('type(vet_notas) = ', type(vet_notas))    #imprime o tipo (ndarray)
print('vet_notas.dtype = ', vet_notas.dtype)     #propriedade "dtype" (float64)
print('vet_notas.ndim = ', vet_notas.ndim)       #propriedade "ndim" (1)
print('vet_notas.shape = ', vet_notas.shape)     #propriedade "shape" (10)
print('vet_notas.data = ', vet_notas.data)       #propriedade "data"
print('vet_notas.strides = ', vet_notas.strides) #propriedade "strides"

#indexação básica
print('-----')
print('primeiro elemento = ', vet_notas[0])
print('último elemento = ', vet_notas[len(vet_notas)-1])
print('3º e 4º elementos = ', vet_notas[2:4])

#modifica a nota do quarto aluno
vet_notas[3] = 9.5
```

27 Neste livro cobriremos outras duas bibliotecas que não fazem parte do Python padrão: ‘pandas’ e ‘Matplotlib’


```
#imprime o novo vetor
print('-----')
print('vet_notas novo = ', vet_notas)
```

Saída [67]:

```
vet_notas = [ 7.8  8.5 10.  9.2  5.  8.5  6.4  8.6  7.5  9. ]
type(vet_notas) = <class 'numpy.ndarray'>
vet_notas.dtype = float64
vet_notas.ndim = 1
vet_notas.shape = (10,)
vet_notas.data = <memory at 0x094AA030>
vet_notas.strides = (8,)
-----
primeiro elemento = 7.8
último elemento = 9.0
3o e 4o elementos = [10.  9.2]
-----
vet_notas novo = [ 7.8  8.5 10.  9.5  5.  8.5  6.4  8.6  7.5  9.]
```

A seguir, algumas explicações:

- O **tipo de um array** ‘NumPy’ é sempre “**numpy.ndarray**”.
- Já o **dtype (tipo dos elementos)** do array) é neste exemplo “float64”. Ele corresponde a um tipo numérico que a ‘NumPy’ utiliza para armazenar números reais com dupla precisão (8 bytes ou 64 bits). É um tipo similar ao tipo básico “float” do Python. A ‘NumPy’ trabalha com diversos outros tipos de dados, conforme veremos na Lição 43.
- Assim como ocorre com as listas, o primeiro elemento de um ndarray de n células possui o índice 0 e o último $n-1$.
- As operações de indexação e fatiamento sobre vetores numpy também funcionam do mesmo jeitinho que ocorre com as listas. Ou seja, você usa colchetes “[]” para indexar e dois pontos “:” para fatiar.

Programa 68 – Carregando um Vetor a Partir de um Arquivo Texto. Suponha que, desta vez, as notas médias dos alunos de Python estejam gravadas em um arquivo chamado “NOTAS_FINAIS.TXT”, cujo conteúdo é reproduzido abaixo:

```
MEDIAS
7.8
8.5
10.0
9.2
5.0
8.5
6.4
8.6
7.5
9.0
```

O programa a seguir mostra que a importação desse arquivo é feita de forma trivial com o uso do método `genfromtxt()` da ‘NumPy’. No exemplo, o parâmetro `skip_header=1` é usado para indicar que desejamos pular a primeira linha. Ou seja, ele é necessário sempre que o arquivo a ser importado possuir um cabeçalho.

#P068: Populando um vetor a partir de um arquivo texto

```
import numpy as np

vet_notas = np.genfromtxt('C:/CursoPython/NOTAS_FINAIS.txt', skip_header=1)

print('vet_notas = ', vet_notas)           #imprime o vetor
print('type(vet_notas) = ', type(vet_notas)) #imprime o tipo (ndarray)
print('vet_notas.dtype = ', vet_notas.dtype) #propriedade "dtype" (float64)
print('vet_notas.ndim = ', vet_notas.ndim)   #propriedade "ndim" (1)
print('vet_notas.shape = ', vet_notas.shape) #propriedade "shape" (10)
print('vet_notas.data = ', vet_notas.data)    #propriedade "data"
print('vet_notas.strides = ', vet_notas.strides) #propriedade "strides"
```

Saída [68]:

```
vet_notas = [ 7.8  8.5 10.  9.2  5.  8.5  6.4  8.6  7.5  9. ]
type(vet_notas) = <class 'numpy.ndarray'>
vet_notas.dtype = float64
vet_notas.ndim = 1
vet_notas.shape = (10,)
vet_notas.data = <memory at 0x094AA100>
vet_notas.strides = (8,)
```

Programa 69 – Outros Métodos para Criar Vetores. Além das maneiras apresentadas nos últimos dois programas, a biblioteca ‘NumPy’ oferece uma série de outros métodos para criar vetores. Estes são apresentados no programa abaixo:

#P069: Miscelânea de métodos para criar vetores NumPy

```
import numpy as np

n1 = np.array([1,2,3]); n2 = np.array([4,5])

#com o método append() podemos combinar vetores
n3 = np.append(n1, n2)
print(n3)           #[1 2 3 4 5]

#arange(): versão NumPy da função range() do Python
a1 = np.arange(11)      #seq. de 0 a 10
a2 = np.arange(0, 16, 5) #seq. de 0 a 15, com 5 como incremento
a3 = np.arange(5, 0, -1) #seq. de 5 a 1

print('-----')
print('a1 = ', a1)      #[0,1,2,3,4,5,6,7,8,9,10]
print('a2 = ', a2)      #[0, 5, 10, 15]
print('a3 = ', a3)      #[5, 4, 3, 2, 1]

#repeat(): gera sequência de valores repetidos
#ones(): gera sequência de valores 1
#zeros(): gera sequência de valores 0
rep1 = np.repeat(100,5)
o1 = np.ones(5)
z1 = np.zeros(5)
```

```

print('-----')
print('rep1 = ', rep1)    #[100 100 100 100 100]
print('o1 = ', o1)       #[1. 1. 1. 1. 1.]
print('z1 = ', z1)       #[0. 0. 0. 0. 0.]

#linspace(): gera elementos "uniformemente espaçados" entre o
#             início e o fim da sequência (neste caso, o valor final é incluído)
ls1 = np.linspace(1,3,9)    #seq. de 1 a 3, com 9 elementos
ls2 = np.linspace(0,2.0/3,4) #seq. de 0 a 2/3, com 4 elementos

print('-----')
print("ls1 = ", ls1) #[1. 1.25 1.5 1.75 2. 2.25 2.5 2.75 3.]
print("ls2 = ", ls2) #[0., 0.22222222, 0.44444444, 0.66666667]

```

Saída [69]:

```

[1 2 3 4 5]
-----
a1 = [ 0  1  2  3  4  5  6  7  8  9 10]
a2 = [ 0  5 10 15]
a3 = [5 4 3 2 1]
-----
rep1 = [100 100 100 100 100]
o1 = [1. 1. 1. 1. 1.]
z1 = [0. 0. 0. 0. 0.]
-----
ls1 = [1. 1.25 1.5 1.75 2. 2.25 2.5 2.75 3. ]
ls2 = [0.      0.22222222 0.44444444 0.66666667]

```

O Quadro 23 apresenta um resumo com a sintaxe dos métodos para a criação de vetores. Na Lição 44, veremos que os mesmos métodos podem ser utilizados para a criação de matrizes. E na Lição 49, veremos que existem métodos para a geração de matrizes a partir de sequências de números aleatórios.

Quadro 23 – Métodos para Criação de Vetores

- **array**(lst): cria um vetor a partir da lista lst;
- **append**(lst₁, ..., lst_n): cria um vetor a partir da concatenação do conteúdo de um conjunto de listas passadas como parâmetro (lst₁, ..., lst_n);
- **arange**([início], fim, [incremento]): cria um vetor a partir de uma sequência que pode ser definida com sintaxe idêntica a da função range();
- **repeat**(x, n): cria um vetor de n posições, todas contendo o valor x;
- **ones**(n): cria um vetor de n posições, todas contendo o valor 1;
- **zeros**(n): cria um vetor de n posições, todas contendo o valor 0;
- **linspace**(início, fim, n): cria um vetor de n posições, com elementos uniformemente espaçados entre os valores especificados em início e fim. Os valores especificados em início e fim serão armazenados no primeiro e no último índice do vetor, respectivamente.



Lição 43 – NumPy: Tipos de Dados

A ‘NumPy’ oferece um rico conjunto de tipos de dados (dtypes) que podem ser especificados em situações onde for preciso ter um controle maior sobre a forma de armazenar os dados em memória (por exemplo, quando você estiver trabalhando com um grande volume de dados). Os principais são relacionados no Quadro 24:

Quadro 24 – “dtypes” da NumPy

- **bool_**: valores booleanos;
- **str**: valores string (alfanuméricos);
- **int8**: inteiros com representação em 8 bits. Faixa de valores: -128 a 127;
- **int16**: inteiros com representação em 16 bits. Faixa de valores: -32.768 a 32.767;
- **int32**: inteiros com representação em 32 bits. Faixa de valores: -2.147.483.648 a 2.147.483.647;
- **int64**: inteiros com representação em 64 bits. Faixa de valores: -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807;
- **int_**: apelido para o tipo inteiro *default*, normalmente “int64”;
- **uint8**: inteiros sem sinal com representação em 8 bits. Faixa de valores: 0 a 255;
- **uint16**: inteiros sem sinal com representação em 16 bits. Faixa de valores: 0 a 65.535;
- **uint32**: inteiros sem sinal com representação em 32 bits. Faixa de valores: 0 a 4.294.967.295;
- **uint64**: inteiros sem sinal com representação em 64 bits. Faixa de valores: 0 a 18.446.744.073.709.551.615;
- **float16**: valores reais positivos e negativos, representação utilizando 5 bits para o expoente e 10 para a mantissa (*half-precision*). Faixa de valores aproximada: -2^{16} a 2^{16} ;
- **float32**: valores reais positivos e negativos, representação utilizando 8 bits para o expoente e 23 para a mantissa (*single-precision*). Faixa de valores aproximada: 2^{-128} a 2^{128} ;
- **float64**: valores reais positivos e negativos, representação utilizando 11 bits para o expoente e 52 para a mantissa (*double-precision*). Faixa de valores aproximada: 2^{-1022} a 2^{1023} ;
- **float_**: apelido para “float64”;
- **complex64**: Número complexo representado por dois floats de 32 bits;
- **complex128**: Número complexo representado por dois floats de 64 bits;
- **complex_**: apelido para “complex128”.

O programa a seguir mostra como criar arrays especificando o tipo de seus elementos. Para tal, basta empregar o parâmetro `dtype` seguido de uma constante no formato “np.tipo”, onde o tipo é um dos tipos de dado que acabamos de apresentar. O programa também mostra como é possível utilizar o método `astype()` para converter o `dtype` de um array.

É importante considerar que, em grande parte das situações práticas, a ‘NumPy’ consegue detectar automaticamente o tipo dos dados que estão sendo lançados no array. No entanto, existem situações em que o tipo de dados “float64” é utilizado como padrão quando o `dtype` é omitido,

como ocorre com métodos `np.ones` e `np.zeros`. Neste caso, dependendo do problema a ser resolvido, pode ser interessante fazer a conversão para um tipo inteiro.

Programa 70 – Especificação do ‘ndtype’ de Arrays

```
#P070: Cria array especificando o tipo de seus elementos
import numpy as np

a = np.array([7, 3, 5, -1, 0], dtype=np.int16)
b = np.array([7, 3, 5, -1, 0])
c = np.ones(3, dtype=np.int8)
d = np.ones(3)

print('a = ', a, a.dtype)
print('b = ', b, b.dtype)
print('c = ', c, c.dtype)
print('d = ', d, d.dtype)

#também é possível converter explicitamente um array de um dtype para outro
#(operação conhecida como "cast")
a = a.astype(np.float64)
print('"a" após conversão = ', a, a.dtype)

#um array de strings também pode ser convertido para algum dtype numérico
#(se algum valor não puder ser convertido, ocorrerá um TypeError)
s = np.array(["2.5", "0", "256"])
s = s.astype(np.float16)
print('"s" após conversão = ', s, s.dtype)
```

Saída [70]:

```
a = [ 7  3  5 -1  0] int16
b = [ 7  3  5 -1  0] int32
c = [1 1 1] int8
d = [1. 1. 1.] float64
"a" após conversão = [ 7.  3.  5. -1.  0.] float64
"s" após conversão = [ 2.5  0. 256.] float16
```

Uma observação importante: a chamada a `astype()` sempre resulta na criação de um novo array, contendo uma cópia dos dados do array original convertidos para um novo dtype.



Lição 44 – NumPy: Criação de Matrizes

Programa 71 – Criando uma Matriz com o Método `np.array()`. Agora que aprendemos a criar vetores de várias formas, podemos avançar para a criação de matrizes. O programa a seguir cria uma matriz “m” de 2 linhas e 3 colunas a partir de uma lista 2d com o método `np.array()`.

```
#P071: Olá Matriz NumPy!
import numpy as np

m = np.array([[7,8,9], [10,11,12]]) #cria matriz a partir de lista 2d
print('m = ', m)                  #imprime a matriz
```

```

print('type(m) = ', type(m))           #imprime o tipo (ndarray)

print('m.dtype = ',m.dtype)           #propriedade "dtype" (int32 ou int64)
print('m.ndim = ',m.ndim)             #propriedade "ndim" (2)
print('m.shape = ',m.shape)           #propriedade "shape" (2,3)
print('m.data = ',m.data)             #propriedade "data"
print('m.strides = ',m.strides)       #propriedade "strides"

#indexação e fatiamento básicos
print('-----')
print('m[0,1] = ', m[0,1])             #[8]          => (célula da 1ª linha, 2ª coluna)
print('m[1,:] = ', m[1,:])             #[10 11 12]    => (toda 2ª linha)
print('m[:,2] = ', m[:,2])             #[9 12]       => (toda 3ª coluna)
print('m[-1,-2:] = ', m[-1,-2:])      #[11 12]      => (última linha, 2 últ. colunas)

#modifica o valor da célula [1,2] (2ª linha, 3ª coluna)
m[1,2] = 999
print('-----')
print('m nova = ', m)                 #imprime a nova matriz

```

Saída [71]:

```

m = [[ 7  8  9]
      [10 11 12]]
type(m) = <class 'numpy.ndarray'>
m.dtype = int32
m.ndim = 2
m.shape = (2, 3)
m.data = <memory at 0x05173CD8>
m.strides = (12, 4)

-----
m[0,1] = 8
m[1,:] = [10 11 12]
m[:,2] = [ 9 12]
m[-1,-2:] = [11 12]

-----
m nova = [[ 7  8  9]
           [10 11 999]]

```

Programa 72 – Carregando uma Matriz a Partir de um Arquivo Texto. Abaixo, apresenta-se o conteúdo do arquivo “DISTANCIAS.csv”, que contém 3 linhas e 3 colunas. Considere que este arquivo armazena as distâncias entre 3 cidades, C1, C2 e C3. As distâncias são simétricas, ou seja, a distância de C1 para C2 é igual a de C2 para C1, e assim da mesma forma para todos os outros pares de cidades.

```

C1, C2, C3
0.0, 10.5, 45.0
10.5, 0.0, 33.2
45.0, 33.2, 0.0

```

A seguir, apresenta-se o código necessário para carregar este arquivo em uma matriz denominada “m_dist”. Mais uma vez, será empregado o método `genfromtxt()`, o mesmo que usamos para importar dados para um vetor. Uma diferença importante é que desta vez **é preciso acrescentar o parâmetro `delimiter`** para indicar que os valores estão separados por vírgula (o

separador padrão é espaço em branco). Outra diferença é que o parâmetro `dtype` foi utilizado para forçar o tipo dos elementos como “float16” em vez de “float64”.

```
#P072 Populando uma matriz a partir de um arquivo texto
import numpy as np

#carrega o arquivo para uma matriz
m_dist = np.genfromtxt('C:/CursoPython/DISTANCIAS.csv',
                      skip_header=1, dtype=np.float16, delimiter=',')

#imprime a matriz
print('m_dist: ', m_dist)
```

Saída [72]:

```
m_dist: [[ 0. 10.5 45. ]
 [ 9.9  0. 33.2]
 [44.5 33.  0. ]]
```

Comentários adicionais:

- Caso o arquivo de entrada contenha algum “**lixo**” (algo que não possa ser interpretado como um número), este será **automaticamente convertido para NaN**.
- Uma vez que você tenha realizado todas as operações desejadas em seu vetor ou matriz, poderá salvá-lo em disco com o uso do método `savetxt()`. No exemplo abaixo, o conteúdo da matriz “m” é salvo no arquivo “minha_matriz.csv” (pasta “C:/CursoPython”) utilizando-se o caractere “;” como delimitador:

```
np.savetxt('C:/CursoPython/minha_matriz.csv', m, delimiter=';')
```

- Também é possível armazenar cada **coluna** em um array diferente, utilizando o parâmetro `unpack = True`. No exemplo abaixo, os vetores “v1”, “v2” e “v3” são criados e preenchidos com os dados da primeira, segunda e terceira colunas respectivamente:

```
v1, v2, v3 = np.genfromtxt('C:/CursoPython/DISTANCIAS.csv',
                          skip_header=1,
                          dtype=np.float16,
                          delimiter=',',
                          unpack=True)
```

Programa 73 – Outros Métodos para Criar Matrizes. Os mesmos métodos utilizados para criar vetores podem ser também utilizados para criar matrizes, bastando indicar o número de linhas e colunas desejadas. Outra forma muito empregada na prática é utilizar o método `np.reshape()`, que transforma um vetor em uma matriz (mais uma vez, bastando para tal indicar o número de linhas e colunas desejadas).

```
#P073: Criando matrizes de várias formas
import numpy as np

#método básico: cria um vetor e depois aplica reshape()
m = np.arange(10)           #cria vetor [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
m = m.reshape((5,2))       #transforma em uma matriz 5 x 2
```

```
#utilizando os métodos que já conhecemos:
mz = np.zeros((4,4)) #matriz 4x4 de zeros (default=float)
mu = np.ones((3,2),dtype=np.int16) #matriz 3x2 de 1's (forcei tipo int16)

#identity() -> cria matriz identidade
mi = np.identity(3) #matriz identidade 3x3

print('m = ', m)
print('-----')
print('mz = ', mz)
print('-----')
print('mu = ', mu)
print('-----')
print('mi = ', mi)
```

Saída [73]:

```
m = [[0 1]
      [2 3]
      [4 5]
      [6 7]
      [8 9]]
-----
mz = [[ 0.  0.  0.  0.]
      [ 0.  0.  0.  0.]
      [ 0.  0.  0.  0.]
      [ 0.  0.  0.  0.]]
-----
mu = [[1 1]
      [1 1]
      [1 1]]
-----
mi = [[ 1.  0.  0.]
      [ 0.  1.  0.]
      [ 0.  0.  1.]]
```

Os dois novos métodos apresentados são explicados no Quadro 25.

Quadro 25 – Métodos Específicos para Criação de Matrizes

- **a.reshape(i,j)**: muda o shape de um ndarray “a”, transformando-o em uma estrutura com *i* linhas e *j* colunas. No exemplo apresentado, o método foi utilizado para transformar um vetor em matriz (caso de uso mais comum). Porém, ele também pode ser empregado para mudar o *shape* de uma matriz (ex: transformar uma matriz 4x3 em uma matriz 6x2). Pode ainda ser empregado para alterar o número de dimensões de um ndarray (transformar uma matriz em um array de 3 dimensões ou vice-versa).
- **identity(n)**: cria uma matriz identidade de dimensão *n*.



Lição 45 – NumPy: Iteração

O comando **for** pode ser normalmente utilizado para iterar (percorrer todos os elementos de) vetores e matrizes. No caso de um vetor, a função **len()** pode ser utilizada para a obtenção do número de elementos nele armazenados. Para matrizes, o “truque” é utilizar a propriedade **shape** para descobrir o número de linhas e colunas: **shape[0]** retorna o número de linhas e **shape[1]** o número de colunas.

Programa 74 – Percorrendo as Células de uma Matriz. Este programa trabalha novamente com o arquivo “DISTANCIAS.csv”. Mas desta vez, depois de carregá-lo para uma matriz, inclui uma rotina que realiza a iteração sobre as células da matriz e imprime as distâncias entre cada cidade.

```
#P074: percorrendo as células de uma matriz
import numpy as np

#(1)-carrega o arquivo para uma matriz
m_dist = np.genfromtxt('C:/CursoPython/DISTANCIAS.csv',
                      skip_header=1,
                      dtype=np.float16,
                      delimiter=',')

#(2)-imprime as distâncias entre cada cidade
for i in range(0, m_dist.shape[0]): #percorre as linhas
    for j in range(0, m_dist.shape[1]): #percorre as colunas
        if (i < j):
            msg = 'Dist. de C' + str(i+1) + ' para C' + str(j+1)
            print(msg + " = " + str(m_dist[i,j]))
```

Saída [74]

```
Dist. de C1 para C2 = 10.5
Dist. de C1 para C3 = 45.0
Dist. de C2 para C3 = 33.2
```



Lição 46 – NumPy: Operações Aritméticas

Em grande parte das situações práticas, não precisaremos utilizar o comando **for** e muito menos o **while** para trabalhar com ndarrays. Isto porque a maioria das operações com os arrays ‘NumPy’ pode ser executada através do mecanismo conhecido como **computação vetorizada** (**vectorization**). Neste processo, as operações são realizadas sobre cada elemento do vetor ou matriz automaticamente, sem a necessidade de programar um laço. Alguns exemplos:

- Se x é um vetor, e fazemos $x * 2$, obteremos como resultado um vetor que conterà todos os elementos de x multiplicados por 2.
- Ao efetuarmos uma soma de duas matrizes $m1$ e $m2$ com os **shapes compatíveis**, teremos como resultado uma nova matriz onde o valor da célula de índice $[0,0]$ será igual a $m1[0,0] +$

$m2[0,0]$; o valor da célula de índice $[0,1]$ será $m1[0,1] + m2[0,1]$, e assim sucessivamente (o mesmo vale para subtração, multiplicação ou divisão).

- Se os *shapes* não forem compatíveis, ocorrerá um erro do tipo “ValueError”.

Programa 75 – Operações Aritméticas com Computação Vetorizada

#P075: operações aritméticas com vetores matrizes

```
import numpy as np

#operações entre vetor e escalares
v1 = np.array([0,5,10])
print('v1*2 = ', v1 * 2)           #[0, 10, 20]
print('v1-1 = ', v1 - 1)           #[-1, 4, 9]
print('v1**3 = ', v1 ** 3)          #[0, 125, 1000]

#soma dois vetores com 3 elementos
print('-----')
x = np.array([0,5,10])
y = np.array([1,2,3])
z = x + y
print('z = ', z)                    #[1, 7, 13]

#soma duas matrizes 2x4
m1 = np.ones((2,4),dtype=np.int16)
m2 = np.array([[1,2,3,4],[5,6,7,8]])
print('-----')
print('m1 = ', m1)
print('m2 = ', m2)
print('m1 + m2 = ', m1 + m2)

#subtração um vetor de 4 posições de uma matriz 3 x 4
m3 = np.ones((3,4),dtype=np.int16)
v2 = np.arange(4)
print('-----')
print('m3 = ', m3)
print('v2 = ', v2)
print('m3 - v2 = ', m3 - v2)

#multiplicação de duas matrizes "a" e "b" com shape 4 x 2
#IMPORTANTE: não se trata da multiplicação matricial da matemática!!!!
#Na verdade, c[0,0] receberá a[0,0] * b[0,0]; c[0,1] receberá a[0,1] * b[0,1]
#e assim sucessivamente.
a=np.array([[1,2,3,4],[5,6,7,8]])
b=np.array([0,2,4,6,8,10,12,14])
b=b.reshape(2,4)
c=a*b
print('-----')
print('a = ', a)
print('b = ', b)
print('c = a*b =', c)
```

Saída [75]

```
v1*2 = [ 0 10 20]
v1-1 = [-1  4  9]
v1**3 = [  0 125 1000]
```

```
-----  
z = [ 1  7 13]  
-----
```

```
m1 = [[1 1 1 1]  
      [1 1 1 1]]  
m2 = [[1 2 3 4]  
      [5 6 7 8]]  
m1 + m2 = [[2 3 4 5]  
           [6 7 8 9]]  
-----
```

```
m3 = [[1 1 1 1]  
      [1 1 1 1]  
      [1 1 1 1]]  
v2 = [0 1 2 3]  
m3 - v2 = [[ 1  0 -1 -2]  
          [ 1  0 -1 -2]  
          [ 1  0 -1 -2]]  
-----
```

```
a = [[1 2 3 4]  
     [5 6 7 8]]  
b = [[ 0  2  4  6]  
     [ 8 10 12 14]]  
c = a*b = [[ 0  4 12 24]  
          [40 60 84 112]]
```

Os dois programas a seguir apresentam uma aplicação mais interessante do conceito de computação vetorizada:

- No Programa 76 mostramos como converter um conjunto de valores de temperatura de graus Celsius para Fahrenheit sem precisar programar um laço (compare-o com o Programa 8).
- E no Programa 77, apresenta-se o jeito ‘NumPy’ de calcular o valor da série $H = 1 + (1 / 2) + (1 / 3) + \dots + (1 / N)$, que havíamos solucionado anteriormente no Programa 9 através da implementação de um laço com o comando `while`. O Programa 77 usa a função pré-definida `sum()` para realizar o somatório de todas as células do vetor (ver Lição 48).

Programa 76 – Conversão de Celsius para Fahrenheit sem laço

```
#P076: Vectorization 1  
import numpy as np  
  
#Conversão de Celsius para Fahrenheit sem laço  
print('-----')  
print('* * Celsius para Fahrenheit:')  
c = np.arange(-20, 101, 10, dtype=np.int16) #(-20°C a 100°C, de 10 em 10)  
f = (c * 9 / 5 + 32)    #a fórmula é aplicada a todos os elementos do vetor  
print('Celsius    =',c)  
print('Fahrenheit=',f.astype(np.int16)) #converte int16 ao exibir
```

Saída [76]:

```
* * Celsius para Fahrenheit:  
Celsius    = [-20 -10  0 10 20 30 40 50 60 70 80 90 100]  
Fahrenheit= [ -4  14 32 50 68 86 104 122 140 158 176 194 212]
```

Programa 77 – Computa o valor da série $H = 1 + (1/2) + (1/3) + \dots + (1/N)$.

#P077: Vectorization 2

```
import numpy as np
```

```
N = 5
```

```
numerador = np.ones(N)
```

```
denominador = np.arange(1,N+1)
```

```
H = sum(numerador / denominador)
```

```
print('* * * cálculo de H = 1 + (1 / 2) + (1 / 3) + ... + (1 / N), dado N =',N)
```

```
print('* * * resposta: H = ', H)
```

Saída [77]:

```
* * * cálculo de H = 1 + (1 / 2) + (1 / 3) + ... + (1 / N), dado N = 5
```

```
* * * resposta: H = 2.2833333333333333
```



Lição 47 – NumPy: Fatiamento de Matrizes

47.1 Fatiamento no Estilo Lista

Com a NumPy você pode indexar e fatiar matrizes utilizando a mesma sintaxe do fatiamento de listas 2d. Porém, conquistando a vantagem adicional de obter sub-matrizes a partir de uma matriz, algo que não é possível com as listas 2d. No exemplo da Figura 20, apresentamos o resultado de vários comandos de indexação e fatiamento de uma matriz “m” com 3 linhas e 4 colunas. Abaixo uma breve explicação sobre cada exemplo:

- **m[2,1]:** esse exemplo é trivial, mostrando como indexar uma célula específica. Basta indicar entre colchetes a linha e depois a coluna, separados por vírgula (lembrando que a linha 2 é na verdade a terceira linha e a coluna 1 é na verdade a segunda coluna, já que os índices de linha e coluna começam com 0).
- **m[-2,-1]:** Ao utilizar índices números negativos, nós acessamos a matriz de trás pra frente. Neste exemplo, estamos indexando a penúltima linha (-2 = segunda linha de baixo para cima), última coluna (-1 = primeira coluna da direita para a esquerda).
- **m[2]:** Se especificarmos apenas um valor entre colchetes, a NumPy vai interpretar que se trata de uma linha (recorde que isso também ocorre com as listas 2d). Sendo assim, “m[2]” retorna, em um array, a linha inteira de índice 2.
- **m[:, :3]:** Este exemplo é mais interessante, pois mostra como podemos obter uma submatriz. A técnica consiste em, dentro dos colchetes, utilizar “:” para especificar a fatia de linhas desejadas, depois usar vírgula e depois mais uma vez utilizar “:” para especificar a fatia de colunas desejadas. Desta forma, “m[:, :3]” significa: gerar a sub-matriz com todas as linhas (“:”) e as colunas 0,1,2 (“:3”) de “m”. O resultado é uma matriz 3x3.
- **m[:, -2:]:** Neste caso, obtém-se a sub-matriz contendo todas as linhas (“:”) e as duas últimas colunas de (“-2:”) de “m”. O resultado é uma matriz com *shape* 3x2.

- **m[:2, 1:3]:** Aqui estamos fatiando uma sub-matriz contendo as linhas 0 e 1 (“:2”) e as colunas 1 e 2 (“1:3”). O resultado é uma matriz com *shape* 2x2.

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[1, 2]
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[:, :3]
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[-1, -2]
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[:, -2:]
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[2]
[1]	5	6	7	8	
[2]	9	10	11	12	

	[0]	[1]	[2]	[3]	
[0]	1	2	3	4	m[:2, 1:3]
[1]	5	6	7	8	
[2]	9	10	11	12	

Figura 20. Exemplos de comandos para indexação e fatiamento estilo lista de uma matriz 4 x 3

47.2 Fatiamento no Estilo Range

Também é possível fatiar ndarrays utilizando notação similar à da função **range()** - aquela que utilizamos para criar um laço **for ... range()**. Neste caso, devemos empregar a sintaxe **m[l_i:l_f:l_k, c_i:c_f:c_k]**, onde:

- l_i: índice inicial para linha
- l_f: índice final para linha (não será incluído no resultado)
- l_k: incremento para os índices de linha
- c_i: índice inicial para coluna
- c_f: índice final para coluna (não será incluído no resultado)
- c_k: incremento para os índices de coluna

A Figura 21 mostra dois exemplos, mais uma vez utilizando a matriz *m*:

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

m

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

m[0:3:2, 0:4:2]

	[0]	[1]	[2]	[3]
[0]	1	2	3	4
[1]	5	6	7	8
[2]	9	10	11	12

m[:, 1::2]

Figura 21. Exemplos indexação no estilo range

- **m[0:3:2, 0:4:2]**: a parte antes da vírgula corresponde à indexação *range* das linhas e a que está depois corresponde à indexação *range* das colunas. Sendo assim, “0:3:2” literalmente significa: “começando na linha 0, vá até a linha 2 pulando de 2 em 2 linhas”. De maneira análoga “0:4:2” significa: “a partir da coluna 0, vá até a linha 3 pulando de 2 em 2 colunas”. O resultado é a matriz 2x2 [[1,3], [9,11]].
- **m[:, 1::2]**: aqui, “:” especifica “todas as linhas”. Já “1::2” quer dizer “partir da coluna 1, vá até a última coluna pulando de 2 em 2 colunas”. O resultado é a matriz 3x2 [[2,4], [6,8], [10,12]]..

47.3 Fancy Indexing

Outra técnica para indexar arrays é a *fancy indexing*, onde você pode selecionar células passando uma lista ou um ndarray de inteiros em uma ordem desejada. Observe os exemplos apresentados na Figura 22.

- No primeiro exemplo, `m[[2,0]]` faz com que toda a linha 2 e toda linha 0 sejam selecionadas, gerando uma nova matriz que conterá essas linhas, na ordem especificada.
- No segundo exemplo, `m[[2,0],[1,3]]` gera um **vetor** com os valores armazenados nas células [2,1] e [0,3].
- No último exemplo, `m[[2,0][:,[1,3]]` gera uma **matriz** 2x2 com os valores armazenados nas células [2,1], [2,3], [0,1], [0,3].

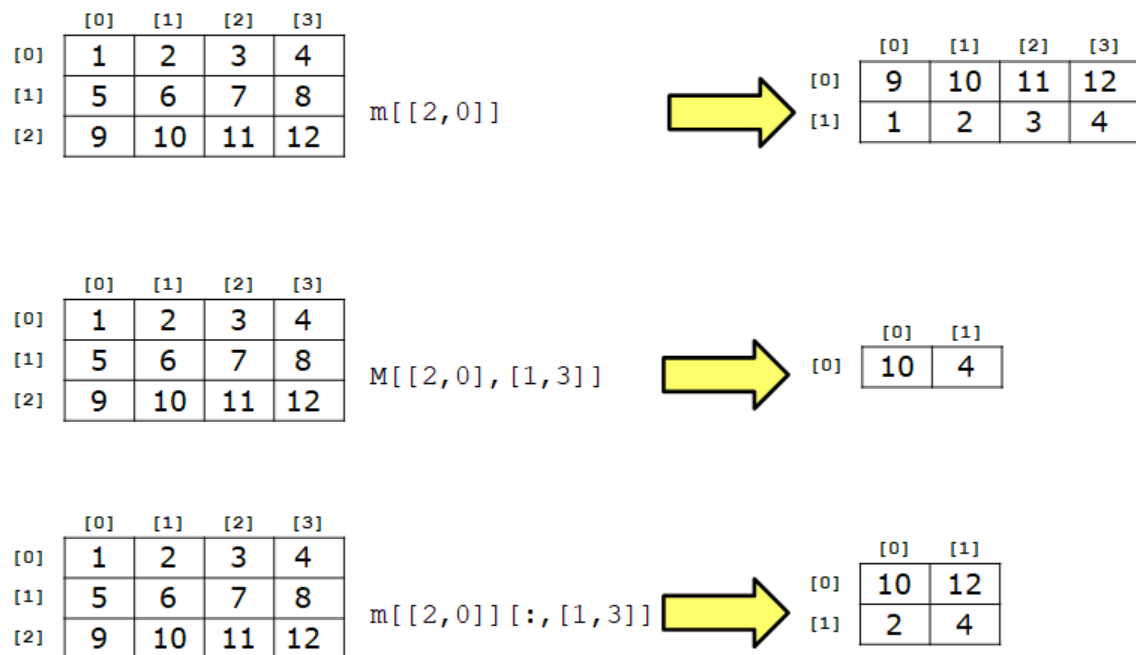


Figura 22. Fancy indexing

Além de ter uma sintaxe bem estranha, o *fancy indexing* possui uma outra diferença em relação ao fatiamento estilo listas ou range: ele produz uma cópia dos dados da matriz, em vez de produzir apenas uma visão. Mais detalhes são fornecidos no Programa 79.

47.4 Indexação Booleana

Por fim, vamos apresentar a indexação booleana. Para isso, vamos retornar ao exemplo das notas dos filmes no IMDb (Lição 15). Mas dessa vez, os dados estarão montados em ndarrays em vez de listas.

Programa 78 – Indexação Booleana. No exemplo a seguir, temos um vetor de strings chamado “vet_filmes” contendo os nomes de 10 filmes e outro vetor de reais chamado “vet_notas” armazenando as notas atribuídas para os mesmos. Mostra-se então como utilizar a indexação booleana para gerar um terceiro vetor chamado “vet_selecionados”, contendo os nomes de todos os filmes cuja nota seja superior a 7.5. Como o seu próprio nome indica, a indexação booleana consiste em selecionar células com base em um teste lógico.

#P078: indexação booleana

`import numpy as np`

```
vet_filmes = np.array(['Procura Insaciável (1971)',
    'Um Estranho no Ninho (1975)',
    'Hair (1979)',
    'Na Época do Ragtime (1981)',
    'Amadeus (1984)',
    'Valmont - Uma História de Seduções (1989)',
    'O Povo Contra Larry Flint (1996)',
    'O Mundo de Andy (1999)',
    'Sombras de Goya (2006)',
    'Dobre placená procházka (2009)'])
```

```
vet_notas = np.array([7.4, 8.7, 7.6, 7.3, 8.3, 7.0, 7.3, 7.4, 6.9, 6.7])

vet_selecionados = vet_filmes[vet_notas > 7.5]

print(vet_selecionados)
```

Saída [78]

```
['Um Estranho no Ninho (1975)' 'Hair (1979)' 'Amadeus (1984)']
```

47.5 Cópia Rasa x Cópia Profunda

Uma importante diferença entre as listas e os ndarrays é o fato de que as fatias extraídas dos ndarrays representam **visões** do array original. Isto significa que, quando você fatia um array, na verdade o Python faz uma cópia rasa²⁸. Desta forma, quaisquer **modificações realizadas na visão serão refletidas no array original**. Essa característica pode parecer estranha, mas lembre-se que a ‘NumPy’ foi projetada para trabalhar com grandes volumes de dados. Em aplicações deste tipo, imagine os problemas de desempenho e memória que ocorreriam se cada fatia realmente copiasse os dados originais!

Felizmente, também é possível gerar uma cópia real dos dados fatiados (isto é, fazer uma cópia profunda) se houver necessidade. Para tal, é preciso utilizar o método **copy()**. O programa a seguir demonstra como fazer o fatiamento através da cópia rasa e da cópia profunda.

Programa 79 – Shallow Copy versus Deep Copy no fatiamento de ndarrays

```
#P079: cópia rasa (shallow copy) x cópia profunda (deep copy)
import numpy as np

v1 = np.arange(1,10)
c1 = v1[0:5]
print('* * * v1 e c1 -> situação inicial')
print('v1=',v1)
print('c1=',c1)

c1[0]=999
print('-----')
print('* * * o fatiamento gera uma visão do array original (shallow copy)')
print('v1 após mudança c1[0]=',v1)
print('c1 após mudança de c1[0]=',c1)

v2 = np.arange(1,10)
c2 = v2[0:5].copy()
print('-----')
print('* * * v2 e c2 -> situação inicial')
print('v2=',v2)
print('c2=',c2)

c2[0]=999
print('-----')
print('* * * o fatiamento com o método copy() faz uma cópia real (deep copy)')
print('v2 após mudança c2[0]=',v2)
print('c2 após mudança de c2[0]=',c2)
```

28 A única exceção é o fatiamento com *fancy indexing*, que gera uma cópia dos dados.

Saída [79]

```
* * * v1 e c1 -> situação inicial
```

```
v1= [1 2 3 4 5 6 7 8 9]
```

```
c1= [1 2 3 4 5]
```

```
-----
```

```
* * * o fatiamento gera uma visão do array original (shallow copy)
```

```
v1 após mudança c1[0]= [999 2 3 4 5 6 7 8 9]
```

```
c1 após mudança de c1[0]= [999 2 3 4 5]
```

```
-----
```

```
* * * v2 e c2 -> situação inicial
```

```
v2= [1 2 3 4 5 6 7 8 9]
```

```
c2= [1 2 3 4 5]
```

```
-----
```

```
* * * o fatiamento com o método copy() faz uma cópia real (deep copy)
```

```
v2 após mudança c2[0]= [1 2 3 4 5 6 7 8 9]
```

```
c2 após mudança de c2[0]= [999 2 3 4 5]
```



Lição 48 – NumPy: Um Pouco de Matemática e Estatística

Diversas funções matemáticas e estatísticas que funcionam de forma vetorizada podem ser aplicadas sobre ndarrays (ou seja, aplicadas sobre todas as células sem a necessidade da programação de um laço). Além disso, podemos gerar arrays a partir de séries de números aleatórios tendo por base diferentes distribuições de probabilidade. Estes são os temas da presente lição.

48.1 Funções Matemáticas

As funções matemáticas são também chamadas de **funções universais** (**ufuncs**) e podem ser classificadas em dois tipos: unárias e binárias. As função unárias, relacionadas no Quadro 26, são aplicadas sobre todos os elementos de **um único array**.

Quadro 26 - Funções Matemáticas Unárias

- `np.abs(a)` : valor absoluto;
- `np.ceil(a)` : arredondamento “pra cima”, ou seja, retorna o menor inteiro com valor igual ou superior ao valor da célula;
- `np.floor(a)` : arredondamento “pra baixo”, ou seja, retorna o maior inteiro com valor igual ou inferior ao valor da célula.
- `np rint(a)` : arredondamento para o inteiro mais próximo, preservando o “dtype”.
- `np.sqrt(a)` : raiz quadrada;
- `np.square(a)` : eleva ao quadrado;
- `np.exp(a)` : computa e^x para cada elemento x do array “a”;
- `np.log(a)` : logaritmo natural (base e);
- `np.log10(a)` : logaritmo na base 10;
- `np.log2(a)` : logaritmo na base 2;
- `np.isnan(a)` : retorna um array booleano que indica, para cada célula de “a”, se a

mesma armazena um valor NaN (True) ou não (False);

- `np.isfinite(a)`: retorna um array booleano que indica, para cada célula de “a”, se a mesma armazena um valor finito (diferente de NaN);
- `np.isfinite(a)`, `np.isinf(a)`: retorna um array booleano que indica, para cada célula de “a”, se a mesma armazena um valor infinito;
- `np.cos(a)`, `np.sin(a)`, `np.tan(a)`, `np.arccos(a)`, `np.arcsin(a)`, `np.arctan(a)`: funções trigonométricas;
- `np.cosh(a)`, `np.sinh(a)`, `np.tanh(a)`, `np.arccosh(a)`, `np.arcsinh(a)`, `np.arctanh(a)`: funções hiperbólicas;

As funções binárias pegam **dois arrays** como entrada e retornam um único como resultado. Quando aplicadas sobre matrizes, as operações são sempre realizadas sobre células de uma mesma posição i,j de cada uma das matrizes de entrada. As funções binárias são relacionadas no Quadro 27.

Quadro 27 - Funções Matemáticas Binárias

- `np.add(a,b)`: adição, isto é soma as células que ocupam posições equivalentes nos arrays “a” e “b”. O mesmo que usar +;
- `np.subtract(a,b)`: subtração. O mesmo que $a - b$;
- `np.multiply(a,b)`: multiplicação. O mesmo que $a * b$; **Importante:** não se trata da multiplicação matricial da matemática, como foi mostrado no Programa 75.
- `np.divide(a,b)`: divisão. O mesmo que a / b ;
- `np.remainder(a,b)`: Resto da divisão inteira de “a” por “b”. O mesmo que $a \% b$;
- `np.power(a,b)`: eleva os elementos no primeiro array às potências indicadas no segundo. O mesmo que $a ** b$;
- `np.equal(a,b)`, `np.not_equal(a,b)`, `np.greater(a,b)`, `np.greater_equal(a,b)`, `np.less(a,b)`, `np.less_equal(a,b)`: executa comparações relacionais entre pares de elementos (um de “a” e outro de “b”), gerando como resultado um array booleano. Mesma coisa que usar `==`, `!=`, `>`, `>=`, `<` e `<=`, respectivamente.
- `np.logical_and(a,b)`, `np.logical_or(a,b)`, `np.logical_xor(a,b)`: executa comparações lógicas (um de “a” e outro de “b”), gerando como resultado um array booleano. Mesma coisa que usar `&`, `|` e `^`, respectivamente.

48.2 Funções Estatísticas

As funções estatísticas podem ser aplicadas a **todos os elementos do array ou a apenas uma de suas *axis* (eixo)**, como mostraremos nos exemplos desta lição. Lembre-se que em um vetor existe apenas a *axis* 0, enquanto em uma matriz existe a *axis* 0 (linha) e a *axis* 1 (coluna). Algumas das funções disponíveis na NumPy são apresentadas no Quadro 28.

Quadro 28 - Funções Estatísticas

- `sum()` : computa a soma de todos os elementos em um array ou de um de seus eixos;
- `mean()` : computa média de todos os elementos em um array ou de seus eixos (células com valor NaN são ignoradas);
- `var()` e `std()` : variância e desvio padrão, com graus de liberdade podendo ser ajustados via parâmetro (o valor *default* é n);
- `min()` e `max()` : menor e maior valor de um array ou de um de seus eixos;
- `argmin()` e `argmax()` : índice do menor e maior valor;
- `cumsum()` : soma cumulativa dos elementos, começando pelo valor 0;
- `cumprod()` : produto acumulado dos elementos, começando pelo valor 1;

A seguir apresentaremos dois exemplos de programas que utilizam as funções estatísticas. No primeiro e mais simples, nosso alvo é um vetor. Já no segundo, onde o alvo é uma matriz, mostramos a forma pela qual podemos trabalhar com um eixo específico.

Programa 80 – Aplicando Funções Estatísticas sobre um Vetor

```
#P080: Funções Estatísticas aplicadas sobre um vetor
import numpy as np

#cria um vetor com as notas de 10 alunos
v = np.array([7.8, 8.5, 10.0, 9.2, 5.0, 8.5, 6.4, 8.6, 7.5, 9.0])

#aplica as funções estatísticas
print('v.sum() = ', v.sum())          #soma dos valores
print('v.max() = ', v.max())          #maior valor
print('v.min() = ', v.min())          #menor valor
print('v.mean() = ', v.mean())        #valor médio
print('np.median(v) = ', np.median(v)) #mediana (observe a sintaxe!)
print('v.std() = ', v.std())          #desvio padrão
print('v.argmax() = ', v.argmax())    #índice do maior valor
print('v.argmin() = ', v.argmin())    #índice do menor valor
```

Saída [80]

```
v.sum() = 80.5
v.max() = 10.0
v.min() = 5.0
v.mean() = 8.05
np.median(v) = 8.5
v.std() = 1.382931668593933
v.argmax() = 2
v.argmin() = 4
```

Programa 81 – Aplicando Funções Estatísticas sobre uma Matriz. Para trabalhar com as mesmas funções sobre matrizes, é preciso conhecer o conceito de “axis” (eixo). Basicamente, trata-se de um parâmetro que indica se as agregações devem ser obtidas por coluna (nesse caso, especifica-se axis=0) ou por linha (nesse caso, especifica-se axis=1). Para exemplificar o conceito apresentado, considere o arquivo “NOTAS_PROVAS.csv”, que armazena as notas de quatro alunos em três diferentes provas. Cada linha corresponde a um aluno e cada coluna a uma prova.

```
P1, P2, P3
9.8, 7.2, 8.0
5.3, 4.0, 3.5
5.5, 8.1, 7.2
7.0, 7.5, 6.5
```

O código abaixo mostra como importar esse arquivo para uma matriz para, em seguida, calcular a média da turma (envolve todas as células, por isso não precisa-se especificar a axis), a média de cada prova (axis=0) e a média de cada aluno (axis=1) .

```
#P081: Estatísticas sobre matrizes + Axis
import numpy as np

#carrega o arquivo "NOTAS_PROVAS.csv" para uma matriz
notas = np.loadtxt('C:/CursoPython/NOTAS_PROVAS.csv',
                  dtype=float,
                  skiprows=1,
                  delimiter=',')

#imprime a matriz e gera as estatísticas
print('notas: ', notas)

print('média geral: ', notas.mean())
print('média de cada prova: ', notas.mean(axis=0))
print('média de cada aluno: ', notas.mean(axis=1))

print('maior nota geral: ', notas.max())
print('maior nota de cada prova: ', notas.max(axis=0))
print('maior nota de cada aluno: ', notas.max(axis=1))
```

Saída [81]

```
notas: [[ 9.8  7.2  8. ]
 [ 5.3  4.   3.5]
 [ 5.5  8.1  7.2]
 [ 7.   7.5  6.5]]
média geral: 6.63333333333
média de cada prova: [ 6.9  6.7  6.3]
média de cada aluno: [ 8.33333333  4.26666667  6.93333333  7.       ]
maior nota geral: 9.8
maior nota de cada prova: [ 9.8  8.1  8. ]
maior nota de cada aluno: [ 9.8  5.3  8.1  7.5]
```

48.3 Operações de Conjunto

A ‘NumPy’ oferece também operações básicas de conjunto que podem ser executadas apenas sobre vetores (Quadro 29).

Quadro 29 - Funções para Conjuntos

- `np.unique(v)`: retorna os valores únicos de um vetor, de forma ordenada;
- `np.intersect1d(v1, v2)`: computa a interseção de `v1` e `v2`. Elementos repetidos são removidos e resultado é retornado de forma ordenada;
- `np.union1d(v1, v2)`: computa a união de `v1` e `v2`. Elementos repetidos são removidos e resultado é retornado de forma ordenada;
- `np.in1d(v1, v2)`: gera um vetor booleano indicando se cada elemento de `v1` pertence a `v2`;
- `np.setdiff1d(v1, v2)`: computa a diferença entre `v1` e `v2`;
- `np.setxor1d(v1, v2)`: computa a diferença simétrica (ou exclusivo) de `v1` e `v2`.

Programa 82 – Operações de Conjunto

#P082: operações de conjunto

```
import numpy as np
```

```
v1 = np.array(["Fisher", "Pearson", "Turing", "Ada"])
```

```
v2 = np.array(["Fisher", "Fisher", "Ada", "Ada"])
```

```
print(np.unique(v2))           #['Ada', 'Fisher']
print(np.in1d(v1, v2))         #[True, False, False, True]
print(np.intersect1d(v1, v2))  #['Ada', 'Fisher']
print(np.union1d(v1, v2))      #['Ada', 'Fisher', 'Pearson', 'Turing']
print(np.setxor1d(v1, v2))     #['Pearson', 'Turing']
```

Saída [82]

```
['Ada' 'Fisher']
[ True False False  True]
['Ada' 'Fisher']
['Ada' 'Fisher' 'Pearson' 'Turing']
['Pearson' 'Turing']
```

48.4 Números Aleatórios e Distribuições de Probabilidade

O módulo `'numpy.random()'` complementa o tradicional pacote `'random'` da *standard library*, com rotinas para gerar arrays de números pseudoaleatórios com base em diferentes tipos de distribuições de probabilidade.

Programa 83 – Módulo `'numpy.random'` (demonstração)

#P083: módulo `'numpy.random'`

```
#np.rand(m): produz números aleatórios uniformemente
#             distribuídos considerando a faixa de 0 a m
import numpy as np
```

```
np.random.seed(293467)           #estabelece a semente
r1 = np.random.rand(3)           #3 núm. aleat., intervalo (0..1], dist. uniforme
r2 = np.random.randint(5,10)     #um inteiro aleat., intervalo (5, 10]
```

```
print('-----')
```

```

print('r1 = ', r1)           #[ 0.84505198  0.59317749  0.11727527]
print('r2 = ', r2)           #8

#A biblioteca NumPy também possui geradores para diversas outras
#distribuições além da distribuição uniforme. Por exemplo:
#binomial, multinomial, Poisson, ...

#Abaixo, um exemplo que gera números, considerando a
#Distribuição Normal padrão ( $\mu=0$ ,  $\sigma=1$ )
r3 = np.random.normal(size=5)
print('-----')
print('r3 = ', r3)

```

Saída [83]

```

r1 = [0.84505198 0.59317749 0.11727527]
r2 = 8
-----
r3 = [ 0.88535299 -0.5022784  1.7012042 -0.04097655 -2.01385894]

```

O Quadro 30 apresenta uma relação contendo a lista parcial dos métodos oferecidos por ‘np.random’. Na notação adotada, considere que “param” é uma lista de parâmetros que pode ser usada para cada tipo de distribuição (consulte a documentação²⁹ da ‘NumPy’ para obter detalhes).

Quadro 30 – Métodos do módulo ‘numpy.random’

- `np.random.seed(n)`: especifica a semente do gerador de números pseudoaleatórios com o valor de *n*;
- `np.random.permutation(seq)`: retorna uma permutação aleatória da sequência *seq* (mas não muda o array original);
- `np.random.shuffle(seq)`: realiza uma permutação aleatória da sequência *seq*, modificando o array original;
- `np.random.rand(range)`: cria um array com o shape especificado em *range* e o preenche com valores aleatórios a partir de uma distribuição uniforme sobre [0,1];
- `np.random.randint(range)`: cria um array de inteiros aleatórios considerando os limites especificados em *range*;
- `np.random.binomial(param)`: extrai um array de uma distribuição Binomial;
- `np.random.normal(param)`: extrai um array de uma distribuição Normal;
- `np.random.beta(param)`: extrai um array de uma distribuição Beta;
- `np.random.chisquare(param)`: extrai um array de uma distribuição Qui-Quadrado;
- `np.random.gamma(param)`: extrai um array de uma distribuição Gamma;
- `np.random.uniform(param)`: extrai um array de uma distribuição Uniforme (0,1];

²⁹ <https://docs.scipy.org/doc/>



Lição 49 – NumPy: Álgebra Linear

O cálculo de determinantes, obtenção da diagonal principal, obtenção da matriz inversa e outras operações úteis da Álgebra Linear também estão disponíveis na 'NumPy', conforme apresenta-se no Quadro 31.

Quadro 31 – Álgebra linear básica

- **diag()** : retorna a diagonal da matriz;
- **det()** : computa o determinante da matriz;
- **inv()** : retorna a inversa de uma matriz quadrada;
- **solve()** : resolve o sistema linear $Ax = b$ para x , onde A é uma matriz quadrada;
- **lstsq()** : calcula a solução de mínimos quadrados para $y = Xb$;
- **dot()** : multiplicação de matrizes;
- **T()** : gera a matriz transposta;
- **eig()** : computa os autovalores e autovetores de uma matriz quadrada;
- **trace()** : computa a soma dos elementos da diagonal;

Programa 84 – Álgebra Linear. Os métodos **T()** ou **transpose()** podem ser utilizados sobre uma matriz “mat” de ordem $m \times n$ para gerar a sua matriz transposta “mat” de ordem invertida $n \times m$. Já o método **dot()** efetua o produto de duas matrizes de ordem $m \times n$ e $n \times p$ (agora sim, é o produto de matrizes da matemática).

```
#P084: Álgebra Linear
import numpy as np

#(1)-Obtendo a Matriz Transposta
print('-----')
m = np.array([0,6,-1,2,5,0])
m = m.reshape((3,2));

print('matriz original 3x2:\n',m);
print('matriz transposta 2x3:\n',m.T);

#(2)-Multiplicação de Matrizes
print('-----')
a = np.array([2,3,1,0,4,5]); a = a.reshape(3,2)
b = np.array([1,2,3,4]); b = b.reshape(2,2)
c = np.dot(a,b)
print('a =\n',a); print('b =\n',b); print('c = a.b =\n',c)
```

In [84]:

matriz original 3x2:

```
[[ 0  6]
 [-1  2]
 [ 5  0]]
matriz transposta 2x3:
[[ 0 -1  5]
 [ 6  2  0]]
```

```
a =
[[2 3]
 [1 0]
 [4 5]]
b =
[[1 2]
 [3 4]]
c = a.b =
[[11 16]
 [ 1  2]
 [19 28]]
```



Lição 50 – NumPy: Ordenação

Programa 85 – Ordenação de Vetores. O método `sort()` ordena os elementos de vetores e matrizes. No segundo caso, você poderá especificar o parâmetro `axis`.

```
#P085: percorrendo as células de uma matriz
import numpy as np
vet_paises = np.array(['Uruguai', 'Brasil',
                       'Chile', 'Costa Rica', 'Equador'])
vet_paises.sort()
print(vet_paises)
```

Saída [85]:
['Brasil' 'Chile ' 'Costa Rica' 'Equador' 'Uruguai']

E assim chegamos ao fim deste capítulo, que introduziu a biblioteca ‘NumPy’, cobrindo suas principais operações, tipos de dados, funções e métodos.

Capítulo VII. Biblioteca 'pandas'

A biblioteca 'pandas' (*Python Data Analysis Library*) é a mais importante e popular biblioteca para ciência de dados do Python. Ela fornece duas estruturas de dados incrivelmente úteis: Series e DataFrames. A primeira serve para estruturar dados de séries temporais, enquanto a segunda foi especialmente projetada para tornar o processo de manipulação de dados tabulares mais rápido, simples e eficaz. As operações sobre estas EDs consistem basicamente em uma combinação das técnicas para processamento de arrays presentes na 'NumPy' com um conjunto de funções específicas para a manipulação de dados tabulares que se assemelham muito às oferecidas pelo Excel e pela linguagem SQL. Alguns exemplos:

- Importar dados estruturados de diferentes tipos de fontes, como arquivos texto, bancos de dados relacionais, planilhas eletrônicas, etc.
- Combinar, de forma inteligente, observações provenientes de diferentes bases de dados (operação conhecida como *merge* ou *join*);
- Produzir resultados agregados e tabulações (*group by*);
- Limpar e transformar bases de dados (ex.: discretização de variáveis, aplicação de filtros sobre linhas e colunas de tabelas, ordenação dos dados, tratamento de dados ausentes, etc.)

Este capítulo apresenta uma breve introdução à biblioteca 'pandas' com ênfase especial nas operações elementares sobre DataFrame.



Lição 51 – pandas: Introdução

A biblioteca pandas oferece duas EDs: **Series** e **DataFrame** (Figura 23).

[BR]	[FR]	[UK]	[IT]	[US]	Series
Real	Euro	Libra	Euro	Dólar	

	<i>nome</i>	<i>idade</i>	<i>salario</i>	<i>uf</i>	DataFrame
P1	Alex	50	5000	RJ	
P2	Carlos	21	2000	RJ	
P3	Jane	55	3500	SP	
P4	Rakesh	37	6500	SP	
P5	Elis	18	2000	RJ	
P6	Isabel	42	4500	MG	
P7	Andres	33	3500	SP	

Figura 23. Estruturas de dados do pacote pandas

- Uma Series nada mais é do que um ndarray unidimensional (vetor) que pode ser indexado através de rótulos em vez de números. Como o próprio nome diz, a principal aplicação prática para as Series é o armazenamento de dados de **séries temporais** (assunto que não faz parte do escopo deste livro). No entanto, ela também possui outras aplicações: na Figura 23, por exemplo, temos uma Series que armazena dados de países. Neste exemplo, o rótulo é a sigla do país e o valor o nome de sua moeda oficial.
- A estrutura DataFrame serve para representar **dados tabulares** similares aos de uma planilha Excel ou de uma tabela de banco de dados relacional. Cada coluna de um DataFrame ‘pandas’ é uma estrutura Series e essas colunas podem possuir tipos diferentes umas das outras (int, float, string, lista, etc.). Este é exatamente o caso do DataFrame mostrado na Figura 23, que mantém informações de pessoas.

Assim como ocorre com a ‘NumPy’, a biblioteca ‘pandas’ não faz parte do Python padrão, mas está incluída em todas distribuições do Python voltadas para ciência de dados. Antes de utilizá-la em um programa, é preciso importá-la através do comando **import**. Neste livro, adotaremos a notação apresentada abaixo para realizar essa tarefa, em que o apelido “pd” é atribuído para a ‘pandas’:

```
import pandas as pd
```

Neste capítulo, apresentaremos uma visão geral da biblioteca ‘pandas’, focando principalmente na importação de diferentes tipos de arquivo e na execução de operações elementares. A ‘pandas’ é considerada a biblioteca mais importante para ciência de dados do Python,

oferecendo um enorme conjunto de ferramentas. É tanta coisa que fica impossível apresentar em um único capítulo! Por isso, decidi escrever um outro livro inteiro que cobre apenas esta biblioteca. O livro se chama “Pandas Python: Data Wrangling para Ciência de Dados” e foi publicado pela editora Casa do Código (<https://www.casadocodigo.com.br/products/livro-pandas-python>).



Lição 52 – pandas: Estrutura Series

Uma Series pode ser entendida como **vetor ‘Numpy’ de dados** associado a um **vetor de rótulos** (também denominado vetor de índices, ou simplesmente índices). Ou seja: Series = vetor de dados + vetor de rótulos.

Programa 86 – Criação e Manipulação de Series. O programa abaixo apresenta duas diferentes maneiras para criar EDs do tipo Series e também as técnicas básicas para manipulação de dados. Os comentários no corpo do programa são utilizados para apresentar explicações.

```
#P086: Olá Series!
import pandas as pd

#(1)-criação de Series especificando os dados (em uma lista),
# mas sem especificar o vetor índices.
# nesse caso os índices serão inteiros de 0 a n-1
# (onde n é o número de elementos da lista)
s1 = pd.Series([70, 30, -1, 50, 0])

#ao imprimir, os índices são mostrados à esquerda e os valores à direita
print('s1:')
print(s1)           #[0:70, 1:30, 2:-1, 3:50, 4:0]

#(2)-indexação, fatiamento e aritmética funcionam como na 'NumPy'
# o detalhe IMPORTANTE é que o mapeamento índice-valor é preservado
print('-----')
print(s1[s1 > 0])    #[0:70, 1:30, 3:50]
print('\n')
print(s1[2:4]);      #[2:-1, 3:50]
print('\n')
print(s1*2);         #[0:140, 1:60, 2:-2, 3:100, 4:0]

#(3)-criação da Series onde índices=siglas de países e valores=nome da moeda
siglas = ['BR', 'FR', 'UK', 'IT', 'US']
moedas = ['Real', 'Euro', 'Libra', 'Euro', 'Dólar']
s2 = pd.Series(moedas, index=siglas)

print('-----')
print('s2:')
print(s2)

#(4)-podemos utilizar os rótulos para indexar
print('-----')
print(s2['UK'])       #'Libra'
print('\n')
print(s2[['BR', 'IT']])  #['BR': 'Real', 'IT': 'Euro']
print('\n')
```

```

print(s2[1:3])           #['FR': 'Euro', 'UK': 'Libra']
print('\n')
print('BR' in s2)        #True
print('AR' in s2)        #False

#(5)-Insere Portugal e Japão e Remove os Estados Unidos e França
print('-----')
s2['PRT'] = 'Euro'
s2['JPN'] = 'Iene'
s2 = s2.drop(labels=['US', 'FR'])
print(s2)

#(6)-Os métodos values e index retornam, respectivamente, os valores
#     e índices da Series, respectivamente
print('-----')
print(s2.values)         #['Real' 'Libra' 'Euro' 'Euro' 'Iene']
print(s2.index)          #Index(['BR', 'UK', 'IT', 'PT', 'JP'], dtype='object')

```

Saída [86]:

```

s1:
0    70
1    30
2    -1
3    50
4     0
dtype: int64

```

```

-----
0    70
1    30
3    50
dtype: int64

```

```

2    -1
3    50
dtype: int64

```

```

0    140
1     60
2     -2
3    100
4      0
dtype: int64

```

```

-----
s2:
BR    Real
FR    Euro
UK    Libra
IT    Euro
US    Dólar
dtype: object
FR    Euro
UK    Libra
dtype: object

```

```

True
False
-----
BR    Real
UK    Libra
IT    Euro
PRT   Euro
JPN   Iene
dtype: object
-----
['Real' 'Libra' 'Euro' 'Euro' 'Iene']
Index(['BR', 'UK', 'IT', 'PRT', 'JPN'], dtype='object')
-----
Libra

BR    Real
IT    Euro
dtype: object

```

Series e Dicionários

- Uma Series também pode ser vista como um dicionário, uma vez que ela consiste em um mapeamento entre chaves e valores. Desta forma, não é surpreendente que a 'pandas' permita com que uma Series possa ser automaticamente criada a partir de um dicionário, como mostrado no exemplo abaixo:

```

dic1 = {'BR': 'Real', 'FR': 'Euro', 'UK': 'Libra', 'IT': 'Euro',
        'US': 'Dólar'}
s2 = pd.Series(dic1)

```

Programa 87 – Alinhamento Automático e Alteração de Índices por Atribuição. Uma característica interessante da Series é o **alinhamento automático** de dados que possuem índices diferentes quando da execução de operações aritméticas. Outra funcionalidade útil é a possibilidade de alterar todos os índices de uma vez utilizando uma atribuição simples. Essas funcionalidades são demonstradas no programa abaixo:

```

#P087: Series - alinhamento automático + alteração de índices por atribuição
import pandas as pd

#(1)-Alinhamento automático
s1 = pd.Series([10,20,30],index=['A','B','C'])
s2 = pd.Series([1,2,3,4],index=['A','B','C','D'])
s3 = s1+s2
print(s3)

#(2)-Alteração de índices
print('-----')

```

```
s3.index = ['i','j','k','l'] #muda os índices de ['A','B','C','D']
                        #para ['i','j','k','l']
```

```
print(s3)
```

Saída [87]:

```
A    11.0
B    22.0
C    33.0
D     NaN
dtype: float64
```

```
-----
i    11.0
j    22.0
k    33.0
l     NaN
dtype: float64
```

Na primeira parte do programa, veja que o índice "D" existe apenas em "s2". Sendo assim, quando o cálculo s1+s2 é realizado, não há como casar "D" com nenhum índice de "s1". Sempre que isso acontece, o valor NaN é automaticamente lançado no resultado da operação. Na segunda parte do programa, mostramos um exemplo de código que troca os índices 'A', 'B', 'C' e 'D' da série "s3" por 'i', 'j', 'k', 'l', respectivamente.

Programa 88 – Importação de Arquivo CSV e uso de Funções Estatísticas. É possível carregar uma ED Series a partir de um arquivo CSV com o uso do método `read_csv()`. Considere, por exemplo, o arquivo "temperaturas.csv" onde cada linha representa um determinado dia do ano acompanhado da temperatura máxima registrada.

```
dia;temp_max
10/02/2019;31
11/02/2019;35
12/02/2019;34
13/02/2019;28
14/02/2019;27
15/02/2019;27
16/02/2019;24
```

O programa a seguir carrega o arquivo "temperaturas.csv" e obtém algumas estatísticas sobre a variável "temp_max".

```
#P088: Series - carregando a partir de CSV e
#      computando estatísticas sobre valores
import pandas as pd
```

```
#(1)-carrega o CSV para uma Series
temperaturas = pd.read_csv('C:/CursoPython/temperaturas.csv',
                           sep=';',
                           index_col = 0,
                           squeeze = True)
```

```
#(2)-imprime os dados
print(temperaturas)
```

```
print(type(temperaturas)) #Para confirmar que foi mesmo lido como Series

#(3)-computa estatísticas do mesmo jeito que faríamos com um ndarray
print('valor máximo=',temperaturas.max())
print('valor mínimo=',temperaturas.min())
print('valor médio=',temperaturas.mean())
```

Saída [88]:

```
dia
10/02/2019    31
11/02/2019    35
12/02/2019    34
13/02/2019    28
14/02/2019    27
15/02/2019    27
16/02/2019    24
Name: temp_max, dtype: int64
<class 'pandas.core.series.Series'>
valor máximo= 35
valor mínimo= 24
valor médio= 29.428571428571427
```

Veja que para importar o arquivo para uma Series, foi preciso fazer uso de três parâmetros no método `read_csv()`:

- `sep=';':` para indicar que o separador é ";"
- `index_col=0:` para indicar que a primeira coluna ("dia") deve ser usada como índice.
- `squeeze=True:` para retornar uma Series em vez de um DataFrame, já que, por padrão, o método `read_csv()` sempre retorna um DataFrame.



Lição 53 – pandas: Introdução à Estrutura DataFrame

Não é exagero afirmar que a ED mais importante para ciência de dados é o DataFrame, que serve para representar dados tabulares em memória, isto é, dados dispostos em **linhas** e **colunas** (ambas podendo ser indexadas). Os DataFrames 'pandas' possuem um rico conjunto de métodos para pré-processamento, filtragem, integração e exploração de bases de dados. Tipicamente, os dados dos DataFrames são obtidos a partir de arquivos. No entanto, nesta lição começaremos mostrando como criar um DataFrame cujo conteúdo é definido dentro do próprio código Python.

Programa 89 – Criação e Manipulação de DataFrames. A seguir, apresenta-se o código de um programa que cria o DataFrame “pessoas”, apresentado na Figura 23, realizando duas operações básicas sobre o mesmo: atribuição de nomes para os índices e colunas (métodos `index` e `column`) e indexação (fique atento aos métodos `iloc` e `loc`).

```
#P089: Olá DataFrame
import pandas as pd

#(1)-Há várias maneiras de construir DataFrames.
# No exemplo abaixo, o fazemos a partir de um dicionário de listas
dados = {'nome': ['Alex', 'Carlos', 'Jane', 'Rakesh', 'Elis', 'Isabel', 'Andres'],
```

```

        'idade': [50,21,55,37,18,42,33],
        'sal': [5000,2000,3500,6500,2000,4500,3500],
        'uf': ['RJ','RJ','SP','SP','RJ','MG','SP']}

pessoas = pd.DataFrame(dados)

print('(1)-DataFrame original\n')
print(pessoas)          #imprime todo o DataFrame

#(2)-Atribui nomes para os índices (método index)
#     e modifica os nomes das colunas (método columns)
#     (agora sim vai ficar com o formato igual ao da Figura 22)
print('-----')
print('(2)-DataFrame com novos nomes de índices e colunas\n')
pessoas.index = ['P1','P2','P3','P4','P5','P6','P7']
pessoas.columns = ['nome','idade','salario','uf']
print(pessoas)

#(3)-INDEXAÇÃO
#[3.1]-Uma LINHA inteira pode ser recuperada pelo seu rótulo ou posição,
#       através dos métodos "loc" e "iloc", respectivamente
print('-----')
print('(3)-Indexação de linhas, colunas e células\n')
print(pessoas.loc['P5'])    #recupera a linha com rótulo 'P5' (Elis)
print('\n')
print(pessoas.iloc[0])     #recupera a linha na posição 0 (primeira linha - Alex)

#[3.2]-Uma COLUMNA inteira pode ser recuperada como uma Series
#       utilizando a notação estilo "dicionário" ou estilo "atributo".
#       A Series retornada terá o mesmo índice do DataFrame
print('-----')
print(pessoas['idade'])     #recupera a coluna "idade" (notação "dicionário")
print('\n')
print(pessoas.nome)        #recupera a coluna "nome" (notação "atributo")

#[3.3]-Uma CÉLULA pode ser recuperada de diferentes formas.
#       Abaixo, utilizamos os métodos "iloc" e "loc" novamente
print('-----')
print(pessoas.iloc[4][2])   #posição da linha + posição da coluna
print(pessoas.iloc[4]['salario']) #posição da linha + rótulo da coluna
print(pessoas.loc['P5'][2])  #rótulo da linha + posição da coluna
print(pessoas.loc['P5']['salario']) #rótulo da linha + rótulo da coluna

```

Saída [89]:

```

nome idade sal uf
(1)-DataFrame original

```

```

    nome idade sal uf
0  Alex   50 5000 RJ
1 Carlos  21 2000 RJ
2  Jane   55 3500 SP
3 Rakesh  37 6500 SP
4  Elis   18 2000 RJ
5 Isabel  42 4500 MG
6 Andres  33 3500 SP

```

```

-----
(2)-DataFrame com novos nomes de índices e colunas

```


	nome	idade	salario	uf
P1	Alex	50	5000	RJ
P2	Carlos	21	2000	RJ
P3	Jane	55	3500	SP
P4	Rakesh	37	6500	SP
P5	Elis	18	2000	RJ
P6	Isabel	42	4500	MG
P7	Andres	33	3500	SP

(3)-Indexação de linhas, colunas e células

```
nome      Elis
idade     18
salario   2000
uf        RJ
Name: P5, dtype: object
```

```
nome      Alex
idade     50
salario   5000
uf        RJ
Name: P1, dtype: object
```

```
-----
P1    50
P2    21
P3    55
P4    37
P5    18
P6    42
P7    33
Name: idade, dtype: int64
```

```
P1    Alex
P2    Carlos
P3    Jane
P4    Rakesh
P5    Elis
P6    Isabel
P7    Andres
Name: nome, dtype: object
```

```
-----
2000
2000
2000
2000
```

index, column e object

- O DataFrame pode ser visto como um dicionário de Series, onde tanto as linhas como as colunas são indexadas. Com relação ao vocabulário adotado pela 'pandas', é importante mencionar que o termo **“index”** é sempre utilizado para índices das **linhas**, enquanto o termo **“column”** é utilizado para os índices das **colunas**.

- Outro detalhe sobre vocabulário, é que o **dtype** de colunas string é sempre apresentado como "object" (como foi mostrado no programa anterior).

No Quadro 32, apresentamos um resumo das opções disponíveis para indexação de linhas, colunas e células de DataFrames (considere que "d" é o nome de um DataFrame em memória):

Quadro 32. Sintaxe da operação de indexação de DataFrames - situações práticas mais comuns

- `d['col']` ou `d.col`: retorna a coluna de nome 'col' (toda a coluna). O resultado é um objeto do tipo Series;
- `d.iloc[:,j]`: retorna a coluna que ocupa a posição *j* (toda a coluna, lembrando que a primeira coluna está na posição 0, a segunda na posição 1, etc.). O resultado é um objeto do tipo Series;
- `d.loc['idx']`: retorna a linha de índice de rótulo 'idx' (linha inteira);
- `d.iloc[i]`: retorna a linha que ocupa a posição *i* (lembrando que a primeira linha está na posição 0, a segunda na posição 1, etc.);
- `d.iloc[i][j]`: retorna o valor da célula que ocupa a linha *i*, coluna *j*;
- `d.iloc[i]['col']`: retorna o valor da célula que ocupa a linha *i*, coluna denominada 'col';
- `d.loc['idx'][j]`: retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna *j*;
- `d.loc['idx']['col']`: retorna o valor da célula que ocupa a linha do índice de rótulo 'idx', coluna denominada 'col'.
- `d[d.col > k]`: **indexação booleana**. Neste exemplo, retorna todas as linhas onde a coluna "d.col" possua valor superior a *k*. Porém, qualquer outro teste sobre uma **única coluna** pode ser montado utilizando-se os operadores relacionais ==, <>, >, >=, <, <=.



Lição 54 – pandas: Importando Arquivos para Data Frames

Esta seção cobre a importação de arquivos estruturados em diversos tipos de formato.

54.1 – Arquivos CSV

Vamos começar abordando os métodos `read_csv()` e `read_table()`. Ambos podem ser utilizados para a leitura de arquivos baseados em caracteres delimitadores, como arquivos CSV. A única diferença entre eles é que o primeiro método tem a vírgula "," como separador padrão, enquanto o segundo utiliza a tabulação ("\t").

Programa 90 – Importação da base de dados "pessoas.csv". Esta base armazena o nome, idade, salário e UF de residência de um conjunto de pessoas.

```
nome, idade, salario, uf
```

```
Alex,50,5000,RJ
Carlos,21,2000,RJ
Jane,55,3500,SP
Rakesh,37,6500,SP
Elis,18,2000,RJ
Isabel,42,4500,MG
Andres,33,3500,SP
```

```
#P090: Importação de CSV padrão para DataFrame
import pandas as pd
pessoas = pd.read_csv('C:/CursoPython/pessoas.csv')
print(pessoas)
```

Saída [90]:

```
   nome  idade  salario uf
0  Alex     50    5000  RJ
1  Carlos   21    2000  RJ
2  Jane     55    3500  SP
3  Rakesh   37    6500  SP
4  Elis     18    2000  RJ
5  Isabel   42    4500  MG
6  Andres   33    3500  SP
```

Simples não? Nesse caso, o programa ficou bem pequeno e nenhum parâmetro precisou ser especificado porque o formato do arquivo de entrada estava idêntico ao modo *default* da `read_csv()`: os dados separados por vírgula e presença de cabeçalho. No entanto, o método é bem flexível e possui uma série de parâmetros que podem ser utilizados para permitir a importação de arquivos que estejam estruturados de forma diferente. Alguns dos principais são apresentados no Quadro 33.

Quadro 33. Parâmetros dos métodos `read_csv()` e `read_table()`

- **sep**: caractere ou expressão regular utilizada para separar campos em cada linha;
- **skiprows**: número de linhas no início do arquivo que devem ser ignoradas;
- **skip_footer**: número de linhas no final do arquivo que devem ser ignoradas;
- **encoding**: padrão de codificação do arquivo. **Importante**: diferentemente da função `open()` do Python padrão, a codificação *default* da 'pandas' é 'utf-8';
- **header**: número da linha que contém o cabeçalho (*default* = 0). Se não houver cabeçalho, deve-se indicar o valor `header=None` ou passar uma lista de nomes utilizando o parâmetro `names`;
- **names**: permite especificar uma lista para os nomes de colunas;
- **index_col**: permite com que uma das colunas seja transformada em índice das linhas;
- **na_values**: sequência de valores que deverão ser substituídos por NA;
- **thousands**: definição do separador de milhar, por exemplo ",", ou ".";
- **squeeze**: caso o arquivo de entrada possua apenas uma coluna, podemos fazer com que uma Series seja retornada em vez de um DataFrame, bastando para isso especificar `squeeze=True`.

A seguir, serão apresentados diversos exemplos que demonstram a utilização dos parâmetros do Quadro 33. Além disso, serão introduzidas as técnicas para importar arquivos JSON, planilhas Excel e tabelas do SQLite para DataFrames.

Programa 91 – Importação de arquivo separado por ponto e vírgula e sem cabeçalho (parâmetros `sep` e `names`). A base de dados "impressoras.txt" contém informações sobre diferentes impressoras produzidas por um determinado fabricante. Cada impressora é caracterizada por quatro variáveis: “modelo” (número do modelo), “colorida” (1=sim, 0=não), “tipo” (‘laser’ ou ‘ink-jet’) e faixa de preço (‘100-200’, ‘>200’).

```
I001;1;ink-jet;>200
I002;0;laser;100-200
I003;0;ink-jet;100-200
I004;1;ink-jet;100-200
I005;0;laser;>200
I006;1;ink-jet;100-200
I007;1;laser;>200
I008;0;ink-jet;100-200
I009;0;ink-jet;100-200
I010;1;laser;>200
```

O programa a seguir mostra como importar esse arquivo, estabelecendo nomes para a suas colunas.

```
#P091: Importação de arquivo separado por ponto e vírgula e sem cabeçalho
#(Parâmetros "sep" e "names")
import pandas as pd

impressoras = pd.read_csv('C:/CursoPython/impressoras.txt',
                          sep=';',
                          names = ["modelo", "colorida", "tipo", "preco"])

print(impressoras)
```

Saída [91]:

	modelo	colorida	tipo	preco
0	I001	1	ink-jet	>200
1	I002	0	laser	100-200
2	I003	0	ink-jet	100-200
3	I004	1	ink-jet	100-200
4	I005	0	laser	>200
5	I006	1	ink-jet	100-200
6	I007	1	laser	>200
7	I008	0	ink-jet	100-200
8	I009	0	ink-jet	100-200
9	I010	1	laser	>200

Programa 92 – Transformação de coluna em índice de linhas. Você deve ter notado que no arquivo "impressoras.txt", o atributo "modelo" é uma espécie de chave identificadora de cada impressora. Se você quisesse utilizá-lo como índice, bastaria alterar o programa da seguinte forma:

```
#P092: Transformando uma coluna em índice do DataFrame (Parâmetro "index_col")
import pandas as pd

impressoras = pd.read_csv('C:/CursoPython/impressoras.txt',
                          sep=';',
                          names = ["modelo", "colorida", "tipo", "preco"],
                          index_col = "modelo")

print(impressoras)
print('-----')
print(impressoras.loc['I007']) #[1,laser,>200]
```

Saída [92]:

	colorida	tipo	preco
modelo			
I001	1	ink-jet	>200
I002	0	laser	100-200
I003	0	ink-jet	100-200
I004	1	ink-jet	100-200
I005	0	laser	>200
I006	1	ink-jet	100-200
I007	1	laser	>200
I008	0	ink-jet	100-200
I009	0	ink-jet	100-200
I010	1	laser	>200

```
-----
colorida      1
tipo          laser
preco         >200
Name: I007, dtype: object
```

Programa 93 – Especificando uma expressão regular como delimitador. O arquivo "log.txt" armazena informações sobre logins efetuados por usuários de um determinado aplicativo. O arquivo registra a "data" e "hora" do acesso, o "login" do usuário e o "status", um número que indica se o login foi bem-sucedido ou não (1=Sim, 0=Não).

27/01/2019	03:15	Jane	1
27/01/2019	12:20	Aldous	1
27/01/2019	14:59	George	0
27/01/2019	16:37	George	1
27/01/2019	21:25	Thomas	1
28/01/2019	00:01	George	1
28/01/2019	04:30	Jane	0
28/01/2019	17:02	John	1

Observe que as informações não estão separadas de uma maneira muito “bem-comportada”. Por exemplo, entre a "data" e a "hora" temos um espaço; entre a "hora" e o "login" dois espaços; entre o "login" e o "status" um número variável de espaços em branco. Felizmente, esta situação pode ser representada com o uso da expressão regular "\s+".

```
#P093: Utilizando expressão regular para definir um delimitador
import pandas as pd
```

```
log = pd.read_csv('C:/CursoPython/log.txt',
                  sep='\s+',
                  names = ["dia", "hora", "login", "status"])

print(log)
```

Saída [93]:

```
   dia  hora  login  status
0 27/01/2019 03:15   Jane     1
1 27/01/2019 12:20  Aldous     1
2 27/01/2019 14:59  George     0
3 27/01/2019 16:37  George     1
4 27/01/2019 21:25  Thomas     1
5 28/01/2019 00:01  George     1
6 28/01/2019 04:30   Jane     0
7 28/01/2019 17:02   John     1
```

54.2 – Arquivos JSON

Programa 94 – Convertendo JSON para DataFrame. O arquivo "cinema.json", introduzido na Lição 35, armazena informações sobre dois filmes. Seu conteúdo é reproduzido a seguir:

```
[
  {
    "titulo": "JSON x CSV",
    "resumo": "o duelo entre dois formatos para representar informações",
    "ano": 2020,
    "genero": ["aventura", "ação", "ficção"]
  },
  {
    "titulo": "JSON James",
    "resumo": "a história de uma lenda do velho oeste",
    "ano": 2018,
    "genero": ["western"]
  }
]
```

Com a biblioteca 'pandas' é possível transferir o conteúdo de um objeto JSON em memória para um DataFrame de forma trivial. Basta para isso, utilizar o método **construtor**³⁰ DataFrame, passando como parâmetros o objeto JSON e uma lista indicando as chaves que desejamos mapear para colunas em nosso DataFrame.

```
#P094: JSON para DataFrame
import json
import pandas as pd
```

³⁰ Construtor é um método especial utilizado especificamente para criar um objeto em memória. O método construtor normalmente possui o mesmo nome da classe do objeto (no nosso caso, "DataFrame"). Normalmente, as classes dispõem de vários tipos de métodos construtores para permitir com que seus objetos possam ser criados de diferentes formas (ex: a partir de um arquivo JSON, a partir de um XML, a partir de uma tabela de BD, etc.). Embora todos os construtores devam possuir o mesmo nome (nome da classe), eles terão parâmetros de entrada diferentes (podemos passar como entrada um JSON, ou um XML, ou uma tabela, etc.). Maiores detalhes são apresentados no Capítulo IX.

```

#(1)-Importa o arquivo JSON para a memória
nomeArq = 'C:/CursoPython/cinema.json'
with open(nomeArq) as f:
    j_filmes = json.load(f)

#(2)-Transfere a informação para um DataFrame
d_filmes = pd.DataFrame(j_filmes, columns=['titulo', 'resumo', 'ano', 'genero'])

print(d_filmes.columns)
print('-filme 1:')
print(d_filmes.iloc[0])
print('-filme 2:')
print(d_filmes.iloc[1])

```

Saída [94]:

```

Index(['titulo', 'resumo', 'ano', 'genero'], dtype='object')
-filme 1:
titulo                JSON x CSV
resumo    o duelo entre dois formatos para representar i...
ano                2020
genero    [aventura, ação, ficção]
Name: 0, dtype: object
-filme 2:
titulo                JSON James
resumo    a história de uma lenda do velho oeste
ano                2018
genero    [western]
Name: 1, dtype: object

```

54.3 – Planilhas Excel

Programa 95 – Importação de Planilha Excel. O exemplo a seguir mostra o código necessário para importar o arquivo “WEATHER.xls” (Figura 24), que possui 5 variáveis e 14 linhas.

	A	B	C	D	E
1	Outlook	Temp	Humidity	Windy	Class
2	sunny	75	70	yes	Play
3	sunny	80	90	yes	DontPlay
4	sunny	85	85	no	DontPlay
5	sunny	72	95	no	DontPlay
6	sunny	69	70	no	Play
7	overcast	72	90	yes	Play
8	overcast	83	78	no	Play
9	overcast	64	65	yes	Play
10	overcast	81	75	no	Play
11	rain	71	80	yes	DontPlay
12	rain	65	70	yes	DontPlay
13	rain	75	80	no	Play
14	rain	68	80	no	Play
15	rain	70	96	no	Play

Figura 24. Planilha “WEATHER.xls”

```
#P095: importa XLS para DataFrame
import pandas as pd

#importa o data frame
nomeArq = 'C:/CursoPython/WEATHER.xls'
df = pd.read_excel(nomeArq)

#checando a importação
print('index:'); print(df.index); print('-----')
print('columns:'); print(df.columns); print('-----')
print('dtypes:'); print(df.dtypes); print('-----')
print(df)
```

In [95]:

```
index:
RangeIndex(start=0, stop=14, step=1)
-----
columns:
Index(['Outlook', 'Temp', 'Humidity', 'Windy', 'Class'], dtype='object')
-----
dtypes:
Outlook    object
Temp       int64
Humidity   int64
Windy      object
Class      object
dtype: object
-----
```

	Outlook	Temp	Humidity	Windy	Class
0	sunny	75	70	yes	Play
1	sunny	80	90	yes	DontPlay
2	sunny	85	85	no	DontPlay
3	sunny	72	95	no	DontPlay
4	sunny	69	70	no	Play
5	overcast	72	90	yes	Play
6	overcast	83	78	no	Play
7	overcast	64	65	yes	Play
8	overcast	81	75	no	Play
9	rain	71	80	yes	DontPlay
10	rain	65	70	yes	DontPlay
11	rain	75	80	no	Play
12	rain	68	80	no	Play
13	rain	70	96	no	Play

54.4 – Bases de Dados SQLite

É possível transferir os dados obtidos por uma consulta `SELECT` sobre um BD SQLite para um DataFrame. Para tal, devemos usar o método construtor `DataFrame`, passando dois parâmetros: o conjunto de linhas retornadas (utilizando método `fetchall()` do módulo 'sqlite') e uma sequência contendo os nomes das colunas.

Programa 96 – Transferindo Resultado de um Comando SQL SELECT para DataFrame.

```
#P096 pandas x sqlite
import sqlite3
import pandas as pd

#(1)-Conecta com o BD
nomeBD = 'C:/CursoPython/RH.db'
minha_conn = sqlite3.connect(nomeBD)

#(2)-Executa comando SQL
c = minha_conn.cursor()
c.execute('SELECT * FROM Funcionario')

#(2.1)-armazena todos os resultados em memória com o método fetchall()
linhas = c.fetchall()

#(2.2)-pega os nomes das colunas com a propriedade description
nomes_colunas = next(zip(*c.description))

#(3)-Transfere os dados obtidos para um DataFrame
d=pd.DataFrame(linhas,columns=nomes_colunas)

print(d)

minha_conn.close()
```

Saída [96]:

	mat	nome	idade	sexo	id_prof
0	M01	George	58	M	5.0
1	M02	Jane	32	F	3.0
2	M03	Aldous	40	M	3.0
3	M04	Thomas	28	M	1.0
4	M05	Mary	43	F	NaN

Salvando o Conteúdo de DataFrame para um Arquivo

- O método `to_csv()` pode ser utilizado para gravar um arquivo CSV a partir do conteúdo de um DataFrame em memória:

```
peessoas.to_csv("C:\CursoPython\saida.csv", sep="\t", index=False)
```

- No exemplo acima, o arquivo “saida.csv” é gravado a partir do conteúdo do DataFrame “peessoas”. A tabulação “\t” foi escolhida como caractere delimitador (parâmetro `sep`) e o parâmetro `index=False` foi utilizado para evitar com que os rótulos dos índices fossem gravados no arquivo.



Lição 55 – pandas: Transformação de DataFrames – Conceitos Básicos

Uma das vantagens de trabalhar com DataFrames é a facilidade para realizar operações de transformação de dados. Os livros sobre a biblioteca 'pandas' costumam dedicar algumas dezenas de páginas a este tema. No entanto, como não dispomos de tanto espaço neste livro, serão apresentados apenas três exemplos. O primeiro envolve a criação, transformação e remoção de colunas usando o Python padrão. O segundo a exclusão de linhas em função do resultado de um teste lógico. E o último a ordenação de acordo com uma ou mais colunas.

Programa 97 – Transformação de Coluna. Neste exemplo, após importar a base de dados "pessoas.csv" para um DataFrame, a coluna "idade" será transformada "faixa_etaria" (ou seja, "faixa_etaria" será incluída e "idade" removida).

```
nome, idade, salario, uf
Alex, 50, 5000, RJ
Carlos, 21, 2000, RJ
Jane, 55, 3500, SP
Rakesh, 37, 6500, SP
Elis, 18, 2000, RJ
Isabel, 42, 4500, MG
Andres, 33, 3500, SP
```

Importante: a biblioteca 'pandas' oferece métodos sofisticados para transformar variáveis sem a necessidade de utilizar laços. No entanto, como o nosso livro é introdutório, optamos por elaborar um exemplo bem mais simples, baseado na utilização da linguagem Python padrão com a definição de uma função e um "lacinho" **for** para aplicar a transformação linha por linha.

```
#P097: Criando e Removendo Colunas em um DataFrame
#      (usando o Python padrão)
import pandas as pd

#(1)-Define função para converter idade em FaixaEtária
def fx_etaria(idade):
    if (idade < 18) :
        return '<18'
    elif (idade >= 18 and idade < 30) :
        return '18-29'
    elif (idade >= 30 and idade < 40) :
        return '30-39'
    else :
        return '>=40'

#(2)-Importa CSV para o DataFrame
pessoas = pd.read_csv('C:/CursoPython/pessoas.csv')
print('* * DataFrame original:')
print(pessoas)
print('-----')

#(3)-Cria uma lista com todas as faixa etárias
lst_faixa_etaria = []
for i in range(len(pessoas)):
```

```

idade = pessoas.iloc[i]['idade'] #pega a idade da pessoa i
faixa_etaria = fx_etaria(idade) #obtem a faixa etaria da pessoa i
lst_faixa_etaria.append(faixa_etaria) #insere no fim da lista

print('* * Lista de faixa etárias:')
print(lst_faixa_etaria)
print('-----')

#(4)-Cria uma nova coluna no DataFrame, chamada "faixa_etaria"
pessoas['faixa_etaria'] = lst_faixa_etaria
print('* * DataFrame com a coluna "faixa_etaria" inserida:')
print(pessoas)
print('-----')

#(5)-Remove a coluna idade
pessoas = pessoas.drop('idade', axis=1)
print('* * DataFrame com a coluna "idade" removida:')
print(pessoas)
print('-----')

```

Saída [97]:

```

* * DataFrame original:
  nome  idade  salario  uf
0  Alex    50    5000  RJ
1  Carlos  21    2000  RJ
2  Jane    55    3500  SP
3  Rakesh  37    6500  SP
4  Elis    18    2000  RJ
5  Isabel  42    4500  MG
6  Andres  33    3500  SP
-----
* * Lista de faixa etárias:
['>=40', '18-29', '>=40', '30-39', '18-29', '>=40', '30-39']
-----
* * DataFrame com a coluna "faixa_etaria" inserida:
  nome  idade  salario  uf  faixa_etaria
0  Alex    50    5000  RJ    >=40
1  Carlos  21    2000  RJ    18-29
2  Jane    55    3500  SP    >=40
3  Rakesh  37    6500  SP    30-39
4  Elis    18    2000  RJ    18-29
5  Isabel  42    4500  MG    >=40
6  Andres  33    3500  SP    30-39
-----
* * DataFrame com a coluna "idade" removida:
  nome  salario  uf  faixa_etaria
0  Alex    5000  RJ    >=40
1  Carlos    2000  RJ    18-29
2  Jane    3500  SP    >=40
3  Rakesh    6500  SP    30-39
4  Elis    2000  RJ    18-29
5  Isabel    4500  MG    >=40
6  Andres    3500  SP    30-39

```

Explicação:

- No Passo 1, está a definição da função "fx_etaria()", que será utilizada no programa para converter a idade de uma pessoa em uma string que indica a faixa etária da mesma. Nenhuma, novidade, já havíamos apresentado esta mesma função no início do Capítulo II.
- No Passo 2, realiza-se a importação do arquivo "pessoas.csv" para um DataFrame chamado "pessoas".
- O Passo 3 é o mais interessante, pois mostra um exemplo de iteração e indexação sobre o DataFrame. Primeiro foi criada uma lista vazia chamada "lst_faixa_etaria". Em seguida, um montou-se um laço com o comando `for` que varreu o DataFrame de "cabo a rabo". Neste laço, a cada iteração *i*, a idade da pessoa *i* é capturada (com o uso da indexação) e, em seguida, sua faixa etária é computada e armazenada no final de "lst_faixa_etaria". Assim, quando o laço é encerrado, obtém-se uma lista com a faixa etária de todas as pessoas do DataFrame.
- No Passo 4, a coluna "faixa_etaria" é acrescentada ao DataFrame "pessoas". O processo é trivial, consistindo em uma simples atribuição: `pessoas['faixa_etaria'] = lst_faixa_etaria`
- Por fim, no Passo 5, o método `drop()` é utilizado para remover a coluna "idade": `pessoas = pessoas.drop('idade', axis=1)`. Veja que foi preciso especificar dois parâmetros: o nome da coluna e "axis=1", para indicar que desejamos remover uma coluna e não uma linha.

É **importantíssimo** repetir: há métodos mais "espertos" (eficientes) que podem ser utilizados na 'pandas' para conseguir o mesmo resultado, inclusive sem precisar do laço montado no Passo 3. No entanto, como este livro é introdutório, optamos por apresentar uma solução menos eficiente, porém mais intuitiva e didática.

Programa 98 – Removendo Linhas. O exemplo a seguir mostra como remover todas as linhas referentes a pessoas acima de 40 anos:

#P098: Criando e Removendo Colunas em um DataFrame (usando o Python padrão)
import pandas **as** pd

```
# (1)-Importa CSV para o DataFrame
pessoas = pd.read_csv('C:/CursoPython/pessoas.csv')
print('* * DataFrame original:')
print(pessoas)
print('-----')

# (2)-Remove linhas em função de um teste lógico
pessoas = pessoas.drop(pessoas[pessoas.idade>40].index)
print('* * DataFrame sem as pessoas que têm mais de 40 anos:')
print(pessoas)
print('-----')
```

Saída [98]:

```
* * DataFrame original:
  nome idade salario uf
```

```

0 Alex 50 5000 RJ
1 Carlos 21 2000 RJ
2 Jane 55 3500 SP
3 Rakesh 37 6500 SP
4 Elis 18 2000 RJ
5 Isabel 42 4500 MG
6 Andres 33 3500 SP

```

*** * DataFrame sem as pessoas que têm mais de 40 anos:**

```

    nome idade salario uf
1 Carlos 21 2000 RJ
3 Rakesh 37 6500 SP
4 Elis 18 2000 RJ
6 Andres 33 3500 SP

```

Utilizando `peessoas[peessoas.idade>40].index` torna-se possível obter todos os índices do DataFrame onde o valor do teste lógico `peessoas.idade>40` resulta em `True`. Esses índices são então utilizados diretamente pelo método `drop`, que se encarregará de remover as respectivas linhas.

Programa 99 – Ordenação de DataFrame. A ordenação de um DataFrame por uma ou mais colunas pode ser obtida com o uso do método `sort_values()`. A lista de colunas é indicada no parâmetro `by` e a especificação do tipo de ordenação (ascendente ou descendente) é feita no parâmetro opcional `ascending` (sendo que ascendente é *default*). No exemplo a seguir, ordena-se o DataFrame por salário ascendente e idade descendente.

#P099: Ordenando um DataFrame por uma ou mais colunas

```

import pandas as pd

#(1)-Importa CSV para o DataFrame
peessoas = pd.read_csv('C:/CursoPython/peessoas.csv')
print('* * DataFrame original:')
print(peessoas)
print('-----')

#(2)-Ordena por salário e idade (ascendente)
peessoas = pessoas.sort_values(by=['salario','idade'], ascending=[True,False])
print('* * DataFrame ordenado por salário e idade:')
print(peessoas)
print('-----')

```

Saída [99]:

*** * DataFrame original:**

```

    nome idade salario uf
0 Alex 50 5000 RJ
1 Carlos 21 2000 RJ
2 Jane 55 3500 SP
3 Rakesh 37 6500 SP
4 Elis 18 2000 RJ
5 Isabel 42 4500 MG
6 Andres 33 3500 SP

```

* * DataFrame ordenado por salário e idade:

	nome	idade	salario	uf
1	Carlos	21	2000	RJ
4	Elis	18	2000	RJ
2	Jane	55	3500	SP
6	Andres	33	3500	SP
5	Isabel	42	4500	MG
0	Alex	50	5000	RJ
3	Rakesh	37	6500	SP



Lição 56 – pandas: Fatiamento de DataFrames

Programa 100 – Fatiamento de DataFrames. A seguir, alguns exemplos de técnicas que podem ser empregadas para obter fatias de um DataFrame que possui rótulos tanto em seus índices como em suas colunas.

#P100: Fatiamento de DataFrames

```
import numpy as np
import pandas as pd
```

#(1)-Cria um DataFrame com 4 linhas e 5 colunas

```
d = pd.DataFrame(np.arange(20).reshape(4,5),
                 index=['L1','L2','L3','L4'], columns=['A','B','C','D','E'])
print('* * DataFrame "d":')
print(d)
```

#(2)-Atribuição por rótulos

```
print('-----')
print('* * Fatiamento por rótulos com "loc":')
print(d.loc['L2',['B','C']])    #pega a linha 'L2', colunas 'B' e 'C'
print('-----')
print(d.loc[:,['A','D','E']])   #pega todas as linhas, colunas 'A', 'D' e 'E'
print('-----')
print(d.loc[['L1','L4'],:])     #pega todas as colunas das linhas 'L1' e 'L4'
```

#(3)-Atribuição por índices

```
print('-----')
print('* * Fatiamento por índices com "iloc":')
print(d.iloc[0:3,3:5])         #pega as linhas 0 a 2 (L1, L2, L3), colunas 3 e 4 (D, E)
print('-----')
print(d.iloc[:,3:5])           #pega todas as linhas, colunas 3 e 4
print('-----')
print(d.iloc[0:3,:])           #pega todas as colunas, linhas 0 a 2
```

#(4)-Atribuição normal

```
print('-----')
print('* * Atribuição normal com "iat":')
d.iat[3,2] = 700    #muda o valor da linha L4, coluna C para 700
print(d)
```

Saída [100]:

```

* * DataFrame "d":
  A B C D E
L1 0 1 2 3 4
L2 5 6 7 8 9
L3 10 11 12 13 14
L4 15 16 17 18 19
-----
* * Fatiamento por rótulos com "loc":
B    6
C    7
Name: L2, dtype: int32
-----
  A D E
L1 0 3 4
L2 5 8 9
L3 10 13 14
L4 15 18 19
-----
  A B C D E
L1 0 1 2 3 4
L4 15 16 17 18 19
-----
* * Fatiamento por índices com "iloc":
  D E
L1 3 4
L2 8 9
L3 13 14
-----
  D E
L1 3 4
L2 8 9
L3 13 14
L4 18 19
-----
  A B C D E
L1 0 1 2 3 4
L2 5 6 7 8 9
L3 10 11 12 13 14
-----
* * Atribuição normal com "iat":
  A B C D E
L1 0 1 2 3 4
L2 5 6 7 8 9
L3 10 11 12 13 14
L4 15 16 700 18 19

```

Para fatiar usando rótulos utiliza-se o método `loc()` e para fatiar por índices o método `iloc()`. Uma coisa que você deve ter percebido no exemplo é que quando o fatiamento é feito por rótulos, o *endpoint* (limite final) é inclusivo, ou seja, entra no resultado final, diferentemente de tudo que vimos no Python até agora. Isso faz sentido, pois este tipo de fatiamento é normalmente feito com o uso de listas, onde são especificadas as linhas e colunas de interesse. Já no fatiamento por índices, o comportamento é o padrão do Python: o *endpoint* não entra no resultado final. O programa

também mostrou o método `iat()`, que pode ser utilizado para modificar uma célula através da indicação da posição da linha e da coluna.



Lição 57 – pandas: Funções Estatísticas em DataFrames

As EDs da 'pandas' são equipadas com um conjunto de métodos para computar estatísticas descritivas. A maioria destas funções serve para extrair um valor escalar de uma Series (como uma média ou soma) ou de uma coluna ou linha de DataFrame.

Programa 101 – Funções Estatísticas. Considere uma empresa hipotética que realiza palestras sobre as linguagens Java, Python e R. Suponha que a empresa promove de 3 a 5 eventos por mês. As palestras sobre cada linguagem são realizadas no mesmo horário por três professores diferentes. O arquivo “palestras.csv”, listado a seguir, apresenta a audiência de todas as palestras realizadas nos meses de fevereiro e março de 2018. Neste arquivo, a primeira informação é o dia do evento, seguido da quantidade de pessoas que assistiram às palestras de Java, Python e R, respectivamente. Note que no dia 22/03 não houve palestra sobre R.

```
dia,java,python,r
01/02/2018,80,78,82
13/02/2018,72,102,96
25/02/2018,100,115,78
08/03/2018,87,87,79
15/03/2018,97,70,78
22/03/2018,109,81,
30/03/2018,90,78,79
```

Considere que a empresa deseja obter estatísticas básicas a respeito da audiência das palestras em diferentes meses. Para resolver este problema, ela poderia utilizar o código apresentado pelo programa a seguir:

```
#P101: Funções estatísticas em DataFrames
import pandas as pd

#(1)-Importa CSV para o DataFrame
palestras = pd.read_csv('C:/CursoPython/palestras.csv', index_col="dia")
print('* * DataFrame original:')
print(palestras)
print('-----')

#(2)-Computa as estatísticas por coluna
print('* * média de público por linguagem:\n ', palestras.mean(), '\n')
print('* * soma de público por linguagem:\n ', palestras.sum(), '\n')

#(3)-Computa as estatísticas por linha
print('* * média de público por dia:\n ', palestras.mean(axis=1), '\n')
print('* * soma de público por dia:\n ', palestras.sum(axis=1), '\n')
```

Saída [101]:
* * DataFrame original:


```

      java python    r
dia
01/02/2018    80     78 82.0
13/02/2018    72    102 96.0
25/02/2018   100    115 78.0
08/03/2018    87     87 79.0
15/03/2018    97     70 78.0
22/03/2018   109     81 NaN
30/03/2018    90     78 79.0
-----
* * média de público por linguagem:
java      90.714286
python    87.285714
r          82.000000
dtype: float64

* * soma de público por linguagem:
java      635.0
python    611.0
r          492.0
dtype: float64

* * média de público por dia:
dia
01/02/2018    80.000000
13/02/2018    90.000000
25/02/2018    97.666667
08/03/2018    84.333333
15/03/2018    81.666667
22/03/2018    95.000000
30/03/2018    82.333333
dtype: float64

* * soma de público por dia:
dia
01/02/2018    240.0
13/02/2018    270.0
25/02/2018    293.0
08/03/2018    253.0
15/03/2018    245.0
22/03/2018    190.0
30/03/2018    247.0
dtype: float64

```

Explicação:

- Neste programa utilizamos as funções **mean()** e **sum()** para obter a média e soma de **todas** as colunas e linhas da tabela (exceto o dia, que espertamente foi carregado para um índice).
- Se você quiser produzir uma estatística sobre uma coluna ou linha específica, basta indicá-la utilizando a notação de indexação. Por exemplo, para obter a média de público apenas da linguagem Java, basta fazer:

```
palestras['java'].mean().
```

- De maneira análoga, para produzir uma estatística sobre uma faixa de colunas ou linhas, utiliza-se a notação de fatiamento. Por exemplo, para obter a soma do público das linguagens Java e Python, você poderia utilizar o método `loc()`:

```
palestras.loc[:, ['java', 'python']].sum()
```

- Observe que o valor ausente da palestra de R foi automaticamente convertido para NaN quando da importação do arquivo. Os valores NaN são ignorados quando as estatísticas são computadas, a não ser que você especifique o parâmetro `skipna=False`.
- Outro detalhe, é que, por padrão a 'pandas' produz estatísticas por coluna. Se você quiser gerar por linhas, deverá especificar o parâmetro `axis=1` (veja que ele é um parâmetro especificado dentro método).

O Quadro 34 apresenta uma lista contendo alguns dos métodos para computar estatísticas descritivas disponibilizados pela 'pandas' (consulte o manual do pacote para maiores detalhes)

Quadro 34. Alguns métodos para computar estatísticas descritivas

- `count()`: conta o número de valores diferentes de NA;
- `min()`, `max()`: valores mínimo e máximo;
- `argmin()`, `argmax()`: índices onde estão localizados os valores mínimo e máximo;
- `sum()`: soma dos valores;
- `cumsum()`: soma cumulativa;
- `mean()`: média dos valores;
- `median()`: mediana;
- `var()`, `std()`: variância e desvio padrão da amostra;
- `mad()`: desvio absoluto médio (do valor médio);
- `skew()`, `kurt()`: assimetria e curtose da amostra;
- `describe()`: computa um conjunto de estatísticas básicas para uma Series ou para todas colunas numéricas de um DataFrame.



Lição 58 – pandas: Junção e Concatenação de DataFrames

Em algumas situações práticas você poderá precisar combinar dois arquivos contendo dados relacionados em um único DataFrame (ex: pedido com produto, empresa com filial, pessoa com domicílio, paciente com prontuário, funcionário com projeto, etc.). Os arquivos a serem combinados possuirão algum tipo de **relacionamento** efetivado por **uma ou mais colunas em comum** (colunas

ou variáveis de ligação). Esta operação é conhecida como *join* (junção) ou *merge*. Trata-se do mesmo tipo de operação que foi apresentada no Capítulo V, quando mostramos como combinar linhas de tabelas de bancos de dados utilizando a instrução `SELECT` da SQL. Isto não é surpresa, pois, como mencionamos no início desse capítulo, as funcionalidades da 'pandas' tiveram inspiração principal na linguagem SQL e no Excel.

Nesta lição, será apresentada a operação de junção entre um `DataFrame` com dados de EMPRESAS e um `Data Frame` com os dados das unidades locais (UL's) destas empresas³¹. Os `DataFrames` serão montados a partir dos arquivos "EMPRESAS.csv" e "ULS.csv", cujos conteúdos são apresentados a seguir.

O arquivo de empresas possui 3 variáveis: "Raiz do CNPJ", "Razão Social" e "Natureza Jurídica Simplificada" (1=Órgão Público; 2=Empresa Privada).

```
raiz,razao,natjur
11111111,DADOSBR,1
22222222,GLOB,2
33333333,LOJA ALPHA,2
44444444,MERCADO XYZ,2
55555555,PREFEITURA,1
66666666,LIMPSEV,2
88888888,ABG,2
```

O arquivo de UL's possui 4 variáveis: "Raiz do CNPJ", "Sufixo do CNPJ", "Nome Fantasia", "Pessoal Ocupado" (ou seja, total de funcionários).

```
raiz,sufixo,fantasia,po
22222222,0001,GLOB I,20
22222222,0002,GLOB II,5
22222222,0003,GLOB III,15
33333333,0001,LOJA A,8
33333333,0002,LOJA B,4
44444444,0001,PRESIDENCIA,112
44444444,0003,LOJA SP,101
44444444,0004,LOJA MG,48
44444444,0005,LOJA POA,50
44444444,0010,LOJA DF, NaN
55555555,0001,SEDE,1800
66666666,0001,LIMPSEV,12
99999999,0001,NO_NO_NO,0
```

Neste exemplo, a coluna em comum é "Raiz do CNPJ". Observe que há duas empresas sem correspondência no arquivo de UL's (raiz=11111111 e raiz=88888888) e existe uma UL sem correspondência no arquivo de empresas (raiz = 99999999).

Programa 102 – Junção de DataFrames. O programa a seguir, realiza a importação dos dois arquivos para `DataFrames` para, em seguida, combinar estes arquivos utilizando três diferentes abordagens: INNER JOIN, LEFT JOIN e OUTER JOIN.

```
#P102: junção de DataFrames
import pandas as pd
```

31 Uma UL corresponde basicamente a um endereço de atuação da empresa

```

#(1)-importa os CSVs
emp = pd.read_csv('C:/CursoPython/EMPRESAS.csv', encoding='ansi')
ul = pd.read_csv('C:/CursoPython/ULS.csv', encoding='ansi')

print('EMP: '); print(emp)
print('-----')
print('UL: '); print(ul)
print('-----')

#(2)-Combina os Arquivos

#2.1 INNER JOIN
c1 = pd.merge(emp, ul, how='inner', on='raiz')
print('resultado do INNER JOIN: ')
print(c1)
print('-----')

#2.2 LEFT JOIN
c2 = pd.merge(emp, ul, how='left', on='raiz')
print('resultado do LEFT JOIN: ')
print(c2)
print('-----')

#2.3 FULL OUTER JOIN
c3 = pd.merge(emp, ul, how='outer', on='raiz')
print('resultado do FULL OUTER JOIN: ')
print(c3)
print('-----')

```

Saída [102]:

* *

EMP:

	raiz	razao	natjur
0	11111111	DADOSBR	1
1	22222222	GLOB	2
2	33333333	LOJA ALPHA	2
3	44444444	MERCADO XYZ	2
4	55555555	PREFEITURA	1
5	66666666	LIMPSERV	2
6	88888888	ABG	2

UL:

	raiz	sufixo	fantasia	po
0	22222222	1	GLOB I	20
1	22222222	2	GLOB II	5
2	22222222	3	GLOB III	15
3	33333333	1	LOJA A	8
4	33333333	2	LOJA B	4
5	44444444	1	PRESIDENCIA	112
6	44444444	3	LOJA SP	101
7	44444444	4	LOJA MG	48
8	44444444	5	LOJA POA	50
9	44444444	10	LOJA DF	NaN
10	55555555	1	SEDE	1800
11	66666666	1	LIMPSERV	12
12	99999999	1	NO_NO_NO	0

 resultado do INNER JOIN:

	raiz	razao	natjur	sufixo	fantasia	po
0	22222222		GLOB	2	1	GLOB I 20
1	22222222		GLOB	2	2	GLOB II 5
2	22222222		GLOB	2	3	GLOB III 15
3	33333333	LOJA ALPHA		2	1	LOJA A 8
4	33333333	LOJA ALPHA		2	2	LOJA B 4
5	44444444	MERCADO XYZ		2	1	PRESIDENCIA 112
6	44444444	MERCADO XYZ		2	3	LOJA SP 101
7	44444444	MERCADO XYZ		2	4	LOJA MG 48
8	44444444	MERCADO XYZ		2	5	LOJA POA 50
9	44444444	MERCADO XYZ		2	10	LOJA DF NaN
10	55555555	PREFEITURA		1	1	SEDE 1800
11	66666666	LIMPSEV		2	1	LIMPSEV 12

 resultado do LEFT JOIN:

	raiz	razao	natjur	sufixo	fantasia	po
0	11111111	DADOSBR		1	NaN	NaN NaN
1	22222222		GLOB	2	1.0	GLOB I 20
2	22222222		GLOB	2	2.0	GLOB II 5
3	22222222		GLOB	2	3.0	GLOB III 15
4	33333333	LOJA ALPHA		2	1.0	LOJA A 8
5	33333333	LOJA ALPHA		2	2.0	LOJA B 4
6	44444444	MERCADO XYZ		2	1.0	PRESIDENCIA 112
7	44444444	MERCADO XYZ		2	3.0	LOJA SP 101
8	44444444	MERCADO XYZ		2	4.0	LOJA MG 48
9	44444444	MERCADO XYZ		2	5.0	LOJA POA 50
10	44444444	MERCADO XYZ		2	10.0	LOJA DF NaN
11	55555555	PREFEITURA		1	1.0	SEDE 1800
12	66666666	LIMPSEV		2	1.0	LIMPSEV 12
13	88888888	ABG		2	NaN	NaN NaN

 resultado do FULL OUTER JOIN:

	raiz	razao	natjur	sufixo	fantasia	po
0	11111111	DADOSBR		1.0	NaN	NaN NaN
1	22222222		GLOB	2.0	1.0	GLOB I 20
2	22222222		GLOB	2.0	2.0	GLOB II 5
3	22222222		GLOB	2.0	3.0	GLOB III 15
4	33333333	LOJA ALPHA		2.0	1.0	LOJA A 8
5	33333333	LOJA ALPHA		2.0	2.0	LOJA B 4
6	44444444	MERCADO XYZ		2.0	1.0	PRESIDENCIA 112
7	44444444	MERCADO XYZ		2.0	3.0	LOJA SP 101
8	44444444	MERCADO XYZ		2.0	4.0	LOJA MG 48
9	44444444	MERCADO XYZ		2.0	5.0	LOJA POA 50
10	44444444	MERCADO XYZ		2.0	10.0	LOJA DF NaN
11	55555555	PREFEITURA		1.0	1.0	SEDE 1800
12	66666666	LIMPSEV		2.0	1.0	LIMPSEV 12
13	88888888	ABG		2.0	NaN	NaN NaN
14	99999999		NaN	NaN	1.0	NO_NO_NO 0

No passo 1, os arquivos CSV de empresas e ULs foram importados para dois DataFrames, respectivamente chamados de “emp” e “ul”. Como os arquivos possuem condificação ANSI, foi

necessário especificar o parâmetro `encoding='ansi'` para avisar à biblioteca pandas que os arquivos estavam codificados desta forma (a pandas adota o 'utf-8' como codificação padrão). No passo 2, foram gerados três DataFrames, chamados “c1”, “c2” e “c3” a partir da junção dos DataFrames “emp” e “ul” pela coluna de ligação “raiz” (parâmetro `on='raiz'`). O Data Frame “c1” foi gerado com o uso da abordagem INNER JOIN (parâmetro `how='inner'`), “c2” por um LEFT JOIN (parâmetro `how='left'`) e “c3” por um FULL OUTER JOIN (parâmetro `how='outer'`). Estas abordagens são explicadas a seguir:

- INNER JOIN: leva para o DataFrame de destino apenas as linhas que “casam” (*match*) nos dois DataFrames de origem.
- LEFT JOIN: leva para o DataFrame de destino as linhas que “casam” nos dois Data Frames de origem. Se uma ou mais linhas do DataFrame especificado à esquerda (“emp”) não casarem, elas são levadas assim mesmo.
- OUTER JOIN: leva para o DataFrame de destino as linhas que “casam” nos dois DataFrames de origem. As linhas dos dois DataFrames que não casam, também são levadas. Na linguagem SQL, este tipo de junção é conhecida como FULL JOIN ou FULL OUTER JOIN.

Programa 103 – Concatenação de DataFrames. Concatenar é uma operação diferente da junção. Concatenar significa adicionar as linhas de um DataFrame no final de outro DataFrame. Na 'pandas', esta operação pode ser feita com o método `concat()`.

```
#P103: Concatenação de DataFrames
import pandas as pd

#(1)-Cria dois DataFrames, "vendas1" e "vendas2"
vendas1 = pd.DataFrame({'produto':['café','suco','chá'],
                        'mes':['jan','jan','jan'],
                        'quant':[1200,350,245]})

vendas2 = pd.DataFrame({'produto':['café','suco','chá','guaraná'],
                        'mes':['fev','fev','fev','fev'],
                        'quant':[1512,487,300,408]})

vendas = pd.concat([vendas1,vendas2], ignore_index=True)

print(vendas)
```

Saída [103]:

```
  produto mes  quant
0   café jan   1200
1   suco jan    350
2   chá jan    245
3   café fev   1512
4   suco fev    487
5   chá fev    300
6 guaraná fev    408
```

Neste exemplo, o DataFrame "vendas1" contendo 3 linhas foi concatenado com o DataFrame "vendas2" contendo 4 linhas, gerando um DataFrame "vendas" com 7 linhas. O parâmetro `ignore_index=True` foi utilizado para produzir novos índices variando de 0 a 6 em "vendas",

Data Wrangling

- **Data Wrangling** (também chamado de **Data Munging**) consiste no processo de transformar e mapear dados do formato "bruto" para outro formato mais apropriado para análise. Ou seja, é a mesma coisa que pré-processar os dados.
- A biblioteca 'pandas' é bastante sofisticada e muito utilizada para *data wrangling*. Ela possui um rico conjunto de funcionalidades para transformação e limpeza de dados que são apresentadas em maiores detalhes no livro "Pandas Python: Data Wrangling para Ciência de Dados" - <https://www.casadocodigo.com.br/products/livro-pandas-python>.
- No presente livro, nos limitamos a apresentar as funções básicas da biblioteca, com destaque para as que realizam a importação de dados. Mas se você decidir trabalhar pra valer com ciência de dados em Python, é recomendado com que você "mergulhe mais fundo" e realize um estudo abrangente da 'pandas'.



Lição 59 – pandas: Produzindo Resultados Agregados

A biblioteca 'pandas' também oferece um método `groupby()` que funciona de maneira similar ao GROUP BY da linguagem SQL.

Programa 104 – Produzindo resultados agregados. O exemplo a seguir mostra como utilizar o método `groupby()` sobre um DataFrame denominado "projetos" que contém informações sobre projetos conduzidos por uma empresa de reformas de fachada hipotética (o conteúdo deste DataFrame é idêntico ao mostrado na Figura 18, Lição 40).

```
#P104: groupby()
import pandas as pd

#(1)-Cria do DataFrame "projetos"
dados = {'codigo': ['P1', 'P2', 'P3', 'P4', 'P5', 'P6'],
         'tipo': ['A', 'A', 'B', 'A', 'B', 'A'],
         'local': ['RJ', 'DF', 'RJ', 'DF', 'RJ', 'SP'],
         'custo': [500000, 900000, 150000, 1000000, None, 850000]}

projetos = pd.DataFrame(dados)
print(projetos)
print('-----')

#(2)-Gera uma variável "grouped" onde a chave é "tipo" e a medida "custo"
grupo_custo_tipo = projetos['custo'].groupby(projetos['tipo'])

#(3)-Computa agregados a partir da variável gerada
print('- média de custo, por tipo de projeto: ', grupo_custo_tipo.mean())
```

```
print('-----')
print('- tot. de proj c/ custo definido, por tipo:', grupo_custo_tipo.count())
```

Saída [104]:

```
codigo tipo local    custo
0   P1   A   RJ  500000.0
1   P2   A   DF  900000.0
2   P3   B   RJ  150000.0
3   P4   A   DF 1000000.0
4   P5   B   RJ      NaN
5   P6   A   SP  850000.0
```

```
-----
- média de custo, por tipo de projeto: tipo
```

```
A   812500.0
```

```
B   150000.0
```

```
Name: custo, dtype: float64
```

```
-----
- tot. de proj c/ custo definido, por tipo: tipo
```

```
A    4
```

```
B    1
```

```
Name: custo, dtype: int64
```

O segredo do programa está na seguinte linha de código:

```
grupo_custo_tipo = projetos['custo'].groupby(projetos['tipo'])
```

Essa linha é responsável por criar uma variável *grouped* ou **objeto GroupBy** chamado "grupo_custo_tipo". Esta variável não armazena o resultado de nenhum cálculo, mas apenas informações que facilitarão a produção de resultados agregados da coluna "custo" (variável numérica) por "tipo" (variável categórica, denominada pela 'pandas' de variável chave do grupo). A partir de "grupo_custo_tipo" será possível produzir tabulações de "custo" por "tipo". No exemplo apresentado, utilizamos as funções **mean()** e **count()** para obter, respectivamente, uma tabela com a média de custo por projeto e outra com o total de projetos com custo definido por tipo.

Muito parecido com o GROUP BY da SQL, não é? Se você quiser tabular por mais de uma coluna, basta especificar uma lista como parâmetro do método **groupby()**. Por exemplo, para criar uma variável *grouped* cruzando custo por tipo e local, você deve utilizar a sintaxe abaixo:

```
grupo_custo_tipo_local = projetos['custo'].groupby([projetos['tipo'],
projetos['local']])
```



Lição 60 – pandas: Trabalhando com Arquivos Grandes

Nesta lição, apresentaremos algumas ferramentas úteis oferecidas pela 'pandas' para situações em que estejamos lidando com arquivos grandes. Afinal de contas, essa é a situação mais comum na prática! Utilizaremos como exemplo o arquivo 'CENSUS.csv'. (Figura 25) . Ele contém 48.842 observações provenientes do Censo de Washington, 1994. Cada observação corresponde aos dados de uma pessoa. A base de dados possui 9 colunas:

- **age:** idade
- **education:** escolaridade
- **education-num:** anos de estudo
- **marital-status:** estado civil
- **occupation:** nome da profissão
- **race:** raça
- **sex:** sexo
- **hours-per-week:** horas trabalhadas por semana
- **class:** o valor '<=50k' indica renda igual ou inferior à US\$ 50.000,00. O valor '>50k' renda superior à US\$ 50.000,00

No exemplo a seguir, mostramos como os métodos **head()** (exibe apenas as primeiras linhas do DataFrame) e **tail()** (exibe apenas as últimas linhas do DataFrame) podem ser utilizados para facilitar a visualização da base. Mostramos ainda como obter um resumo estatístico simplificado das variáveis numéricas com o uso do método **describe()**.

```

1 age,education,education-num,marital-status,occupation,race,sex,hours-per-week,native-country,income
2 25,11th,7,Never-married,Machine-op-inspct,Black,Male,40,United-States,<=50K
3 38,HS-grad,9,Married-civ-spouse,Farming-fishing,White,Male,50,United-States,<=50K
4 28,Assoc-acdm,12,Married-civ-spouse,Protective-serv,White,Male,40,United-States,>50K
5 44,Some-college,10,Married-civ-spouse,Machine-op-inspct,Black,Male,40,United-States,>50K
6 18,Some-college,10,Never-married,?,White,Female,30,United-States,<=50K
7 34,10th,6,Never-married,Other-service,White,Male,30,United-States,<=50K
8 29,HS-grad,9,Never-married,?,Black,Male,40,United-States,<=50K
9 63,Prof-school,15,Married-civ-spouse,Prof-specialty,White,Male,32,United-States,>50K
10 24,Some-college,10,Never-married,Other-service,White,Female,40,United-States,<=50K
11 55,7th-8th,4,Married-civ-spouse,Craft-repair,White,Male,10,United-States,<=50K
12 65,HS-grad,9,Married-civ-spouse,Machine-op-inspct,White,Male,40,United-States,>50K
13 36,Bachelors,13,Married-civ-spouse,Adm-clerical,White,Male,40,United-States,<=50K
14 26,HS-grad,9,Never-married,Adm-clerical,White,Female,39,United-States,<=50K
15 58,HS-grad,9,Married-civ-spouse,?,White,Male,35,United-States,<=50K
16 48,HS-grad,9,Married-civ-spouse,Machine-op-inspct,White,Male,48,United-States,>50K
17 43,Masters,14,Married-civ-spouse,Exec-managerial,White,Male,50,United-States,>50K
18 20,Some-college,10,Never-married,Other-service,White,Male,25,United-States,<=50K
19 43,HS-grad,9,Married-civ-spouse,Adm-clerical,White,Female,30,United-States,<=50K
20 37,HS-grad,9,Widowed,Machine-op-inspct,White,Female,20,United-States,<=50K
21 40,Doctorate,16,Married-civ-spouse,Prof-specialty,Asian-Pac-Islander,Male,45,?,>50K
22 34,Bachelors,13,Married-civ-spouse,Tech-support,White,Male,47,United-States,>50K
23 34,Some-college,10,Never-married,Other-service,Black,Female,35,United-States,<=50K
24 72,7th-8th,4,Divorced,?,White,Female,6,United-States,<=50K
25 25,Bachelors,13,Never-married,Prof-specialty,White,Male,43,Peru,<=50K
26 25,Bachelors,13,Married-civ-spouse,Prof-specialty,White,Male,40,United-States,<=50K
27 45,HS-grad,9,Married-civ-spouse,Craft-repair,White,Male,90,United-States,>50K
28 22,HS-grad,9,Never-married,Adm-clerical,White,Male,20,United-States,<=50K
29 23,HS-grad,9,Separated,Machine-op-inspct,Black,Male,54,United-States,<=50K
30 54,HS-grad,9,Married-civ-spouse,Craft-repair,White,Male,35,United-States,<=50K
31 32,Some-college,10,Never-married,Prof-specialty,White,Male,60,United-States,<=50K
32 46,Some-college,10,Married-civ-spouse,Exec-managerial,Black,Male,38,United-States,>50K
33 56,11th,7,Widowed,Other-service,White,Female,50,United-States,<=50K
34 24,Bachelors,13,Never-married,Sales,White,Male,50,United-States,<=50K

```

Figura 25. Arquivo "CENSUS.csv"

Programa 105 – Trabalhando com Arquivos Grandes.

#P105: Carregando dados do Censo de Washington para DataFrame

```
import pandas as pd
```

```
#(1)-importa o DataFrame
```

```
nomeArq = 'C:/CursoPython/CENSUS50.csv'
```

```
df = pd.read_csv(nomeArq)
```

```

#(2)-alguns métodos
#imprime as primeiras linhas
print('head():'); print(df.head())
print('-----')

#imprime as últimas linhas
print('tail():'); print(df.tail())
print('-----')

#checando os rótulos das colunas e dos índices
print('index:')
print(df.index)           #inteiro, 0 a 48842 com passo 1

print('columns:')
print(df.columns)         #object: ['age', 'education', ..., 'class']
print('-----')

#checando os tipos das colunas
print('dtypes:'); print(df.dtypes)
print('-----')

#checando os dados na "camada NumPy" (camada "por trás" do DataFrame)
print('values'); print(df.values)
print('-----')

#(3)-método describe()
print('describe():')
print(df.describe())

```

Saída [105]:

```

head():
  age  education  education-num  ...  hours-per-week  native-country  income
0  25      11th           7  ...      40  United-States  <=50K
1  38      HS-grad           9  ...      50  United-States  <=50K
2  28  Assoc-acdm          12  ...      40  United-States  >50K
3  44  Some-college          10  ...      40  United-States  >50K
4  18  Some-college          10  ...      30  United-States  <=50K

```

[5 rows x 10 columns]

```

tail():
  age  education  ...  native-country  income
48837  27  Assoc-acdm  ...  United-States  <=50K
48838  40    HS-grad  ...  United-States  >50K
48839  58    HS-grad  ...  United-States  <=50K
48840  22    HS-grad  ...  United-States  <=50K
48841  52    HS-grad  ...  United-States  >50K

```

[5 rows x 10 columns]

```

index:
RangeIndex(start=0, stop=48842, step=1)
columns:
Index(['age', 'education', 'education-num', 'marital-status', 'occupation',
      'race', 'sex', 'hours-per-week', 'native-country', 'income'],
      dtype='object')
-----

```

```

dtypes:
age          int64
education    object
education-num int64
marital-status object
occupation   object
race         object
sex          object
hours-per-week int64
native-country object
income       object
dtype: object
-----
values
[[25 '11th' 7 ... 40 'United-States' '<=50K']
 [38 'HS-grad' 9 ... 50 'United-States' '<=50K']
 [28 'Assoc-acdm' 12 ... 40 'United-States' '>50K']
 ...
 [58 'HS-grad' 9 ... 40 'United-States' '<=50K']
 [22 'HS-grad' 9 ... 20 'United-States' '<=50K']
 [52 'HS-grad' 9 ... 40 'United-States' '>50K']]
-----
describe():
      age  education-num  hours-per-week
count 48842.000000  48842.000000  48842.000000
mean   38.643585    10.078089    40.422382
std    13.710510     2.570973    12.391444
min     17.000000     1.000000     1.000000
25%     28.000000     9.000000    40.000000
50%     37.000000    10.000000    40.000000
75%     48.000000    12.000000    45.000000
max     90.000000    16.000000    99.000000

```

Capítulo VIII. Biblioteca 'Matplotlib'

Gráficos são ferramentas importantes não apenas para apresentar resultados, mas também quando você está realizando algum tipo de estudo preliminar de uma base de dados. Este capítulo introduz a biblioteca 'Matplotlib', a mais utilizada para produção de gráficos do ambiente Python.

Utilizando a 'Matplotlib' é possível gerar diferentes tipos de gráfico 2D e 3D com qualidade de publicação a partir de poucas linhas de código. Neste capítulo, apresentaremos exemplos que envolvem a produção dos seguintes tipos de gráfico: gráfico de linha, gráfico de barras, diagrama de dispersão, histograma e boxplot. Além disso, o capítulo aborda a utilização do recurso subplot, que permite com que dois ou mais gráficos possam ser desenhados em uma mesma figura.



Lição 61 – Matplotlib: Introdução

Antes de começarmos a criar os nossos gráficos, vale a pena examinar a Figura 26, que apresenta os principais componentes de um gráfico 'Matplotlib'³²:

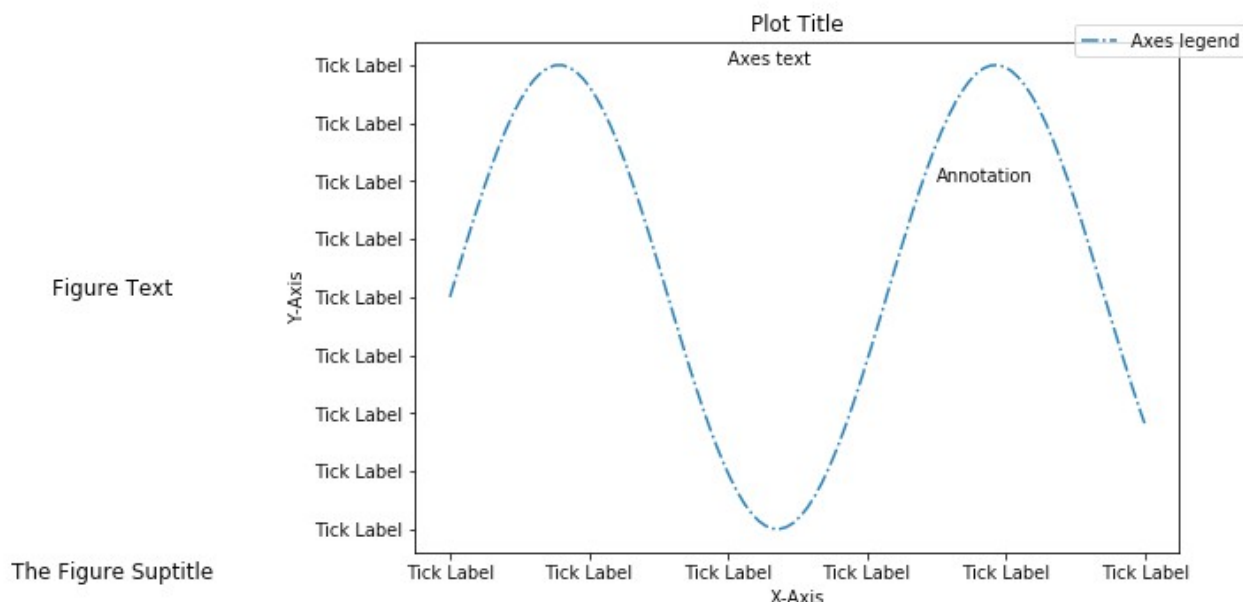


Figura 26. Principais componentes de um gráfico da 'Matplotlib'

Qualquer gráfico possui dois componentes principais: **Figura** e **Eixos** (axes)

- A “Figura” é o componente de mais alto nível. Trata-se da janela completa onde tudo está desenhado. Uma figura pode ter título, subtítulo, entre outras propriedades e é na figura que os “Eixos” são adicionados.
- Os “Eixos” correspondem à área onde os dados são plotados por funções como `plot()`, que mostraremos na próxima lição. Os “Eixos” também tem propriedades, como legendas, limites, etc.

Nas lições a seguir, serão apresentados exemplos de programas para a produção de diferentes tipos de gráfico com a 'Matplotlib'. Os exemplos também mostram como configurar certos parâmetros e propriedades dos gráficos. A importação da biblioteca será feita da seguinte forma:

```
import matplotlib.pyplot as plt
```

O significado é: "importe o submódulo "pyplot" do pacote "matplotlib", renomeando-o para "plt".

³² Figura retirada de: <https://www.datacamp.com/community/tutorials/matplotlib-tutorial-python>



Lição 62 – Matplotlib: Gráfico de Linha

Para criar um gráfico de linha simples tudo que você precisa é de dois ndarrays, um deles com os valores para a coordenada x e o outro para a coordenada y .

Programa 106 – Gráfico de Linha. Este exemplo mostra como importar a biblioteca 'Matplotlib' e utilizar as duas funções básicas para plotar o gráfico da função $y = 2x + 5$, para x variando de 1 a 10. Essas funções são: `plot()` e `show()`. A explicação sobre os comandos é realizada em comentários colocados dentro do código.

```
#P106: Olá Matplotlib!

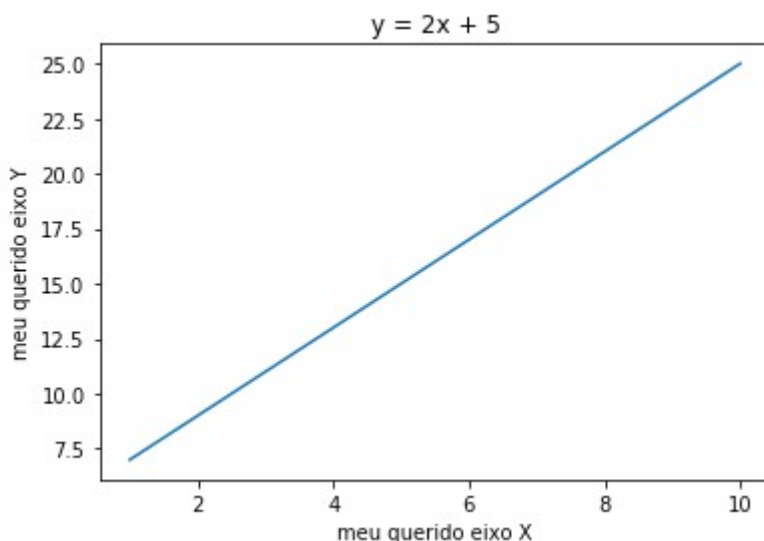
#importa o submódulo "pyplot" do pacote "matplotlib",
#renomeando-o para "plt"
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(1,11)
y = 2 * x + 5

plt.title('y = 2x + 5')           #título do gráfico
plt.xlabel('meu querido eixo X')  #rótulo do eixo X
plt.ylabel('meu querido eixo Y')  #rótulo do eixo Y

plt.plot(x,y)                    #plota os dados
plt.show()                       #mostra o gráfico
```

Saída [106]:





Lição 63 – Matplotlib: Configurações Básicas

Programa 107 – Configurações Básicas de um Gráfico. O arquivo “TX_NATALIDADE.csv” apresenta a taxa bruta de natalidade por mil habitantes no Brasil, entre os anos de 2000 e 2015³³.

```
ano,taxa
2000,20.86
2001,20.28
2002,19.73
2003,19.19
2004,18.66
2005,18.15
2006,17.65
2007,17.18
2008,16.72
2009,16.29
2010,15.88
2011,15.50
2012,15.13
2013,14.79
2014,14.47
2015,14.16
```

O programa a seguir produz um gráfico a partir dos dados deste arquivo, com o “Ano” no eixo *x* e a “Taxa de Natalidade” no eixo *y*, mas dessa vez apresentando diversas novidades em relação ao programa da lição anterior.

```
#P107: Taxa Bruta de Natalidade
import matplotlib.pyplot as plt
import numpy as np

#carrega o arquivo para uma matriz
m = np.loadtxt('C:/CursoPython/TX_NATALIDADE.csv',
               skiprows=1, dtype=float, delimiter=',')

#cria um vetor c/ dados da 1a coluna (Ano) e um outro com os da 2a (Taxa)
x = np.array(m[:,0], dtype=np.int32)
y = m[:,1]

#plota a linha na cor vermelha
plt.plot(x, y, color='red')

#plota os pontos como círculos (parâmetro 'ro'... existem outros) marrons
plt.plot(x, y, 'ro', color='brown')

#define a escala dos eixos X e Y
plt.ylim(min(y)*0.9, max(y)*1.05)
plt.xlim(2000,2015)

#define os "xticks" do eixo X
#(se não colocar fica bem feio, pois ele exhibe 2000.5, 2001.5, ...)
```

³³ Fonte: IBGE, Projeção da População do Brasil - 2013.

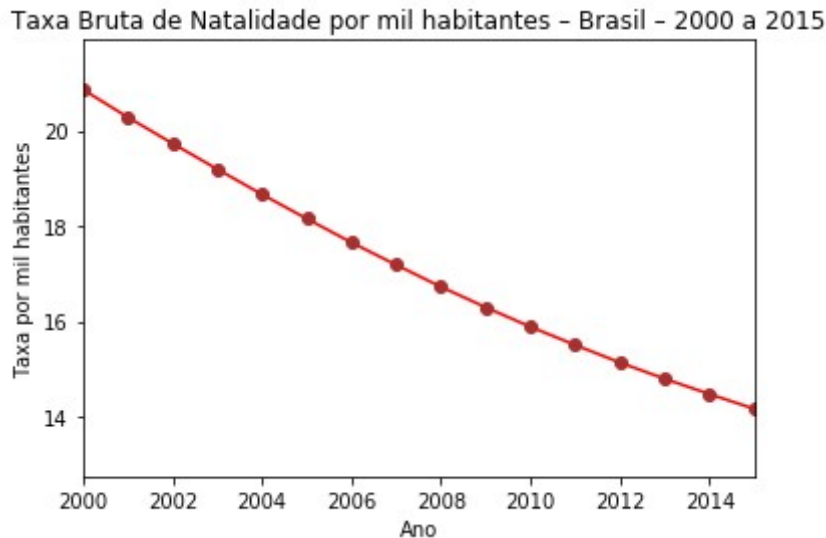
```
plt.xticks(np.arange(2000, 2015, 2))

#define o título do gráfico
plt.title('Taxa Bruta de Natalidade por mil habitantes - Brasil - 2000 a 2015')

#define os rótulos
plt.xlabel('Ano') #rótulo do eixo X
plt.ylabel('Taxa por mil habitantes') #rótulo do eixo Y

#finalmente, mostra o gráfico
plt.show()
```

Saída [107]:



As principais novidades desse programa são:

- Dois “plots” foram utilizados, um para as linhas e outro para os pontos. Cores diferentes foram utilizadas para cada plot.
- Os limites dos eixos x e y foram configurados com os métodos `xlim` e `ylim`, respectivamente. A 'Matplotlib' sempre tenta definir estes limites de forma automática, mas utilizando estes métodos você pode defini-los manualmente. Veja que os limites para o eixo y foram estabelecidos de forma “dinâmica” (considerando o menor e o maior valor da taxa). Já no eixo x , a configuração utilizou valores fixos.
- O método `xticks` foi utilizado para definir com exatidão os valores (`xticks`) a serem exibidos no eixo x . Foi especificado de 2000 a 2015, saltando de 2 em 2.



Lição 64 – Matplotlib: Gráfico de Barras

Nos exemplos anteriores, gráficos foram gerados a partir de arrays 'NumPy'. Desta vez vamos mudar um pouquinho e gerar o gráfico a partir de um DataFrame. Isto não é nada surpreendente,

tendo em vista que cada coluna de um DataFrame é uma Series e uma Series é nada mais do que um ndarray com uma bonita maquiagem!

Programa 108 – Gráfico de Barras. O arquivo “CINTO.csv” armazena os dados consolidados de uma pesquisa feita apenas com pessoas do sexo masculino sobre o uso do cinto de segurança.

```
usa_cinto,porcentagem
Sempre,39.2
Quase Sempre,17.9
Algumas Vezes, 15.8
Raramente, 10.6
Nunca, 16.5
```

O código seguinte mostra como produzir um gráfico de barras simples a partir deste arquivo com o uso do método **bar()**.

```
#P108: Gráfico de Barras
import matplotlib.pyplot as plt
import pandas as pd

#carrega o arquivo para uma matriz
d = pd.read_csv('C:/CursoPython/cinto.csv')

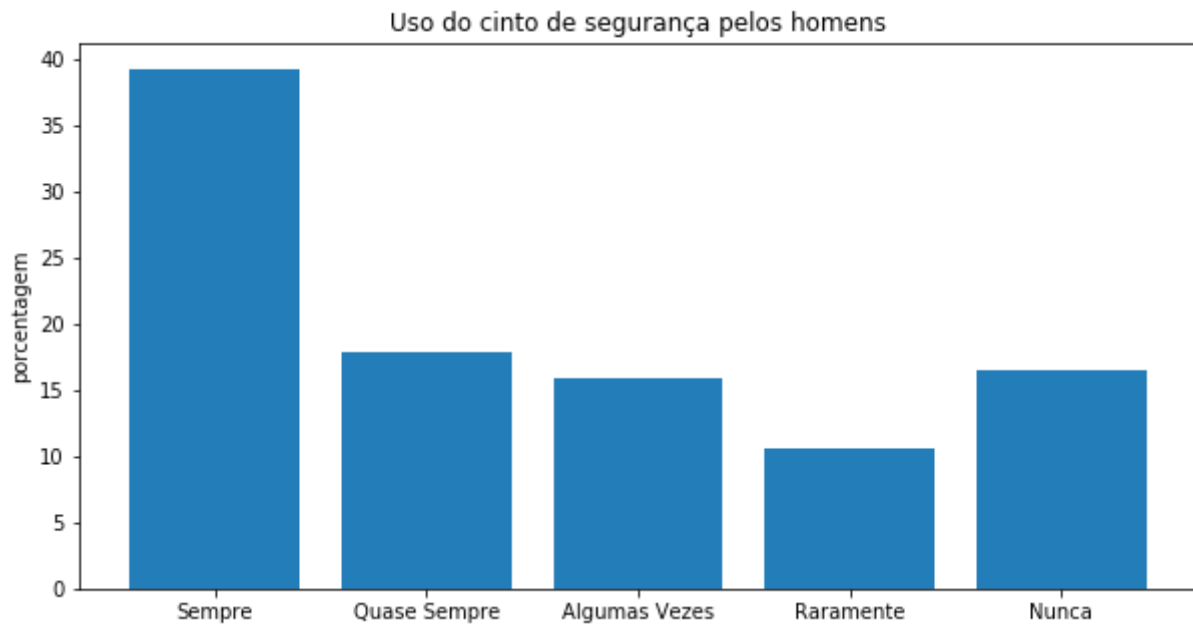
#aumenta o tamanho do gráfico (pois os nomes das categorias são longos)
plt.figure(figsize=(10,5))

#plota com as categorias no eixo X e a porcentagem no eixo Y
plt.bar(d['usa_cinto'], d['porcentagem'])

#associa um label para o Eixo Y e coloca o título no gráfico
plt.ylabel('porcentagem')
plt.title('Uso do cinto de segurança pelos homens')

#mostra o gráfico
plt.show()
```

Saída [108]:



Lição 65 – Matplotlib: Gráfico de Dispersão

Programa 109 – Gráfico de Dispersão. Considere o arquivo “TRIANGLE.csv”, que possui 1000 *data points* (apenas alguns são mostradas abaixo). Este arquivo possui 3 colunas. As duas primeiras, representam as coordenadas do *data point* nos eixos *x* e *y*, respectivamente. Já a terceira coluna, indica a classe a qual o *data point* pertence (0, 1, 2 ou 3)

```
10.6427,13.294,0
12.1989,10.2245,0
10.3681,11.8402,0
10.551,12.4163,0
12.1474,10.3369,0
...
0.683088,2.47889,3
-0.524523,-3.40659,3
```

A seguir, apresenta-se um programa que gera um gráfico de dispersão com os dados do arquivo utilizando um método chamado `scatter()`. Adicionalmente, exemplifica-se o uso do recurso `colormap`, que possibilita com que os pontos pertencentes a cada classe sejam marcados com cores diferentes.

```
#P109: Gráfico de Dispersão
import matplotlib.pyplot as plt
import numpy as np

#carrega o arquivo para uma matriz
m = np.loadtxt('C:/CursoPython/TRIANGLE.csv',
               dtype=float, delimiter=',')
```

```

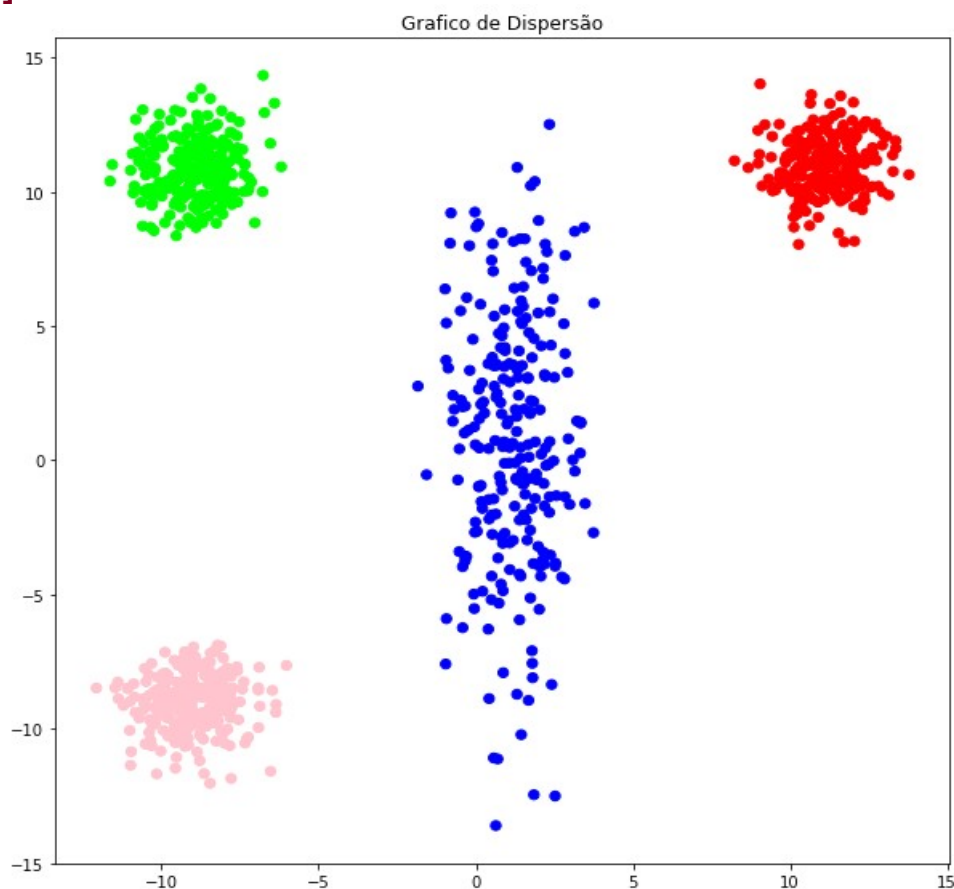
#cria um vetor para cada coluna
x = m[:,0] #eixo x
y = m[:,1] #eixo y

#a classe tem que ser "int", para que o uso do colormap seja possível
classe = np.array(m[:,2],dtype=np.int16)

#configura e plota o gráfico
plt.figure(figsize=(10,10))
colormap = np.array(['red', 'lime', 'pink','blue'])
plt.scatter(x, y, c=colormap[classe], s=40)
plt.title('Grafico de Dispersão')
plt.show()

```

Saída [109]:



Lição 66 – Matplotlib: Histograma

Programa 110 – Histograma. A função `hist()` é utilizada para gerar histogramas. Nesta seção, mostramos como gerar um histograma bem simples a partir do arquivo “granjas.csv”. Considere que ele armazena a produção de ovos de 15 granjas hipotéticas.

```
GRANJA;TOTAL
Granja A;1012
Granja B;5238
Granja C;2136
Granja D;6724
Granja E;7652
Granja F;4910
Granja G;3751
Granja H;4002
Granja I;3151
Granja J;1311
Granja K;2311
Granja L;5697
Granja M;7182
Granja N;4331
Granja O;3506
```

```
#P110: Histograma
```

```
import matplotlib.pyplot as plt
import pandas as pd
```

```
#(1)-carrega o arquivo CSV para um DataFrame
```

```
d = pd.read_csv('C:/CursoPython/GRANJAS.csv', delimiter=';')
```

```
#(2)-carrega o arquivo CSV para um DataFrame
```

```
t = d.iloc[:,1].values
```

```
#(3)-plota o histograma com 8 bins
```

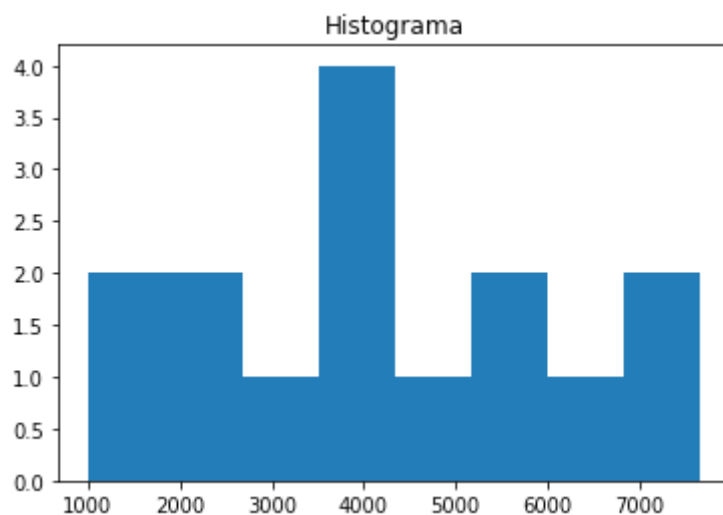
```
num_bins = 8
```

```
plt.hist(t, num_bins)
```

```
plt.title('Histograma')
```

```
plt.show()
```

Saída [110]:





Lição 67 – Matplotlib: Boxplot

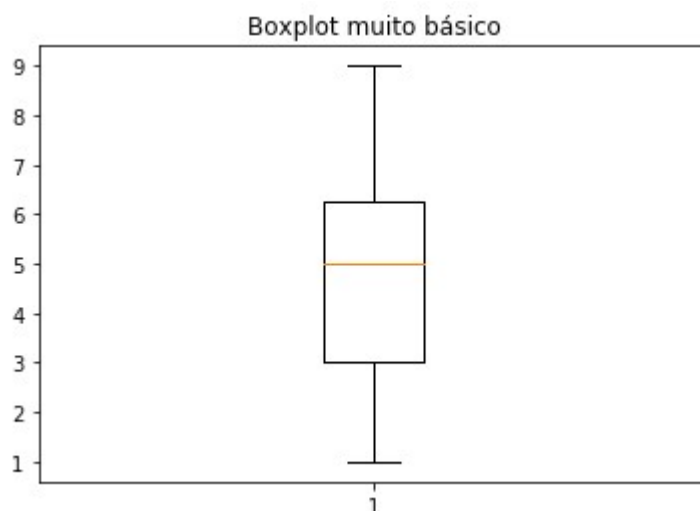
Programa 111 – Boxplot. O boxplot (diagrama de caixa) é uma das ferramentas mais utilizadas para explorar bases de dados. No programa a seguir, apresentamos um boxplot simples, gerado a partir do conjunto de dados: [3, 7, 2, 1, 4, 3, 8, 3, 5, 7, 4, 5, 6, 7, 6, 5, 4, 3, 9, 5].

```
#P111: Boxplot
import matplotlib.pyplot as plt
import numpy as np

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

plt.boxplot(dados)
plt.title('Boxplot muito básico')
plt.show()
```

Saída [111]:



Lição 68 – Matplotlib: Configurando Subplots

A função `subplot()` permite com que diversos gráficos sejam colocados em um *grid* dentro de uma mesma figura.

Programa 112 – Subplots. O exemplo a seguir monta um *grid* com duas colunas e uma linha, exibindo o gráfico da função $y = \text{raiz}(x)$ à esquerda e de $y = \log_2(x)$ à direita, para x variando de 1 a 100. Adicionalmente, o exemplo mostra como definir o tamanho da figura (a janela completa em volta dos dois subplots) e alguns parâmetros de configuração das linhas.

```
#P112: Subplots
import matplotlib.pyplot as plt
```

```
import numpy as np

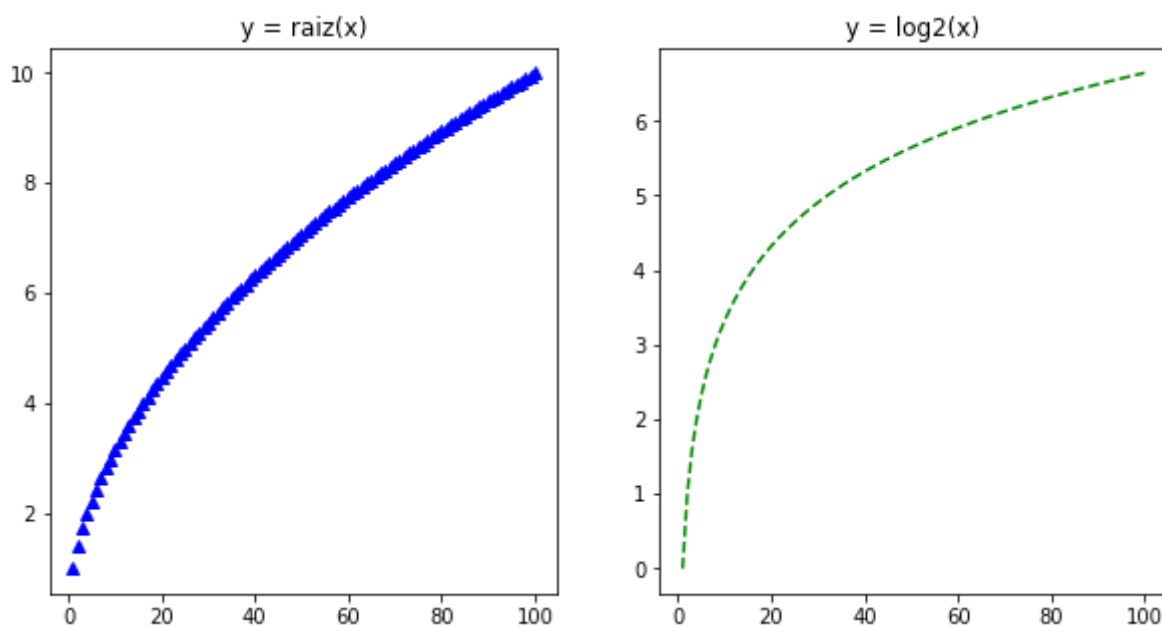
x = np.arange(1,101)
y1 = np.sqrt(x)
y2 = np.log2(x)

#configura o tamanho da figura
#o primeiro parâmetro é a largura e o segundo a altura
fig = plt.figure(figsize=(10,5))

#configura um subplot com 1 linha e 2 colunas
#seta a primeira coluna como ativa para receber o gráfico y1 = raiz(x)
plt.subplot(1,2,1)
plt.plot(x, y1, 'b^') # 'b^' = triângulos azuis
plt.title("y = raiz(x)")

#adiciona o segundo subplot, ativando a coluna 2
plt.subplot(1,2,2)
plt.plot(x, y2, '--g') # 'g--' = linha verde pontilhada
plt.title("y = log2(x)")
```

Saída [112]:



Lição 69 – Matplotlib: Salvando um Gráfico

Programa 113 – Salvando um Gráfico em Arquivo. Gráficos podem ser salvos de maneira simples com o uso do método `savefig()`. No exemplo a seguir, após gerarmos um gráfico, o salvamos com o nome de “figura1.png”.

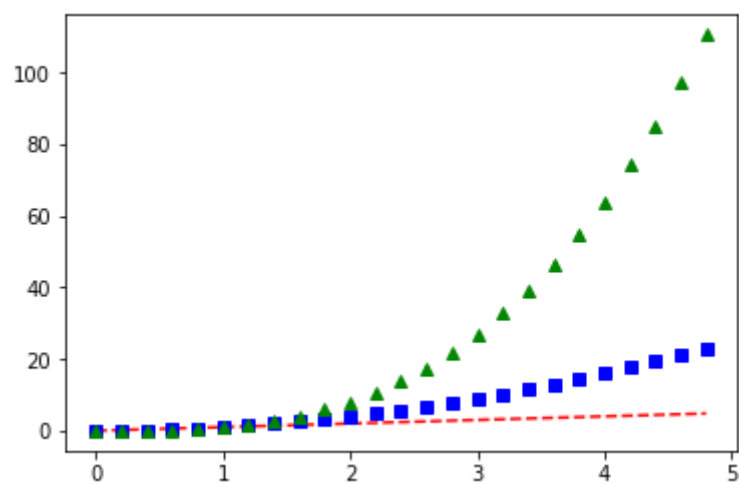
```
#P113: salvando uma figura

#este exemplo está em https://matplotlib.org/users/pyplot\_tutorial.html
#ele desenha três gráficos na mesma figura
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(0., 5., 0.2)
plt.plot(x, x, 'r--', x, x**2, 'bs', x, x**3, 'g^')

#salva em um arquivo "png" com o método savefig()
plt.savefig("C:/CursoPython/figural.png")
```

Saída [113]:



Capítulo IX. Introdução à Programação Orientada a Objetos

Embora o paradigma procedural seja o mais simples para o desenvolvimento de seus *scripts* de ciência de dados, é importante deixar claro que as funcionalidades de programação orientada a objetos (POO) do Python também são bastante úteis em diversas situações práticas. Por exemplo, apenas para que você tenha uma ideia, a grande maioria dos pacotes com implementações de algoritmos de *machine learning* e métodos estatísticos é desenvolvida seguindo a filosofia da orientação a objetos. Isso porque a POO favorece a manutenção, distribuição e tratamento de erros de código, aspectos que precisam receber especial atenção quando desenvolvemos programas que serão utilizados por muitas pessoas (e não apenas em nossos próprios experimentos).

Desta forma, a orientação a objetos também deve estar na lista de tópicos importantes para estudo de qualquer aspirante ao posto de *pythonista* profissional. Afinal de contas, se um dia você precisar examinar o código fonte de um pacote ou se quiser desenvolver o seu próprio pacote, precisará conhecer sobre o assunto. Neste capítulo apresentamos uma breve introdução à POO, focando em seus conceitos elementares: classe, objeto, atributos (ou propriedades), métodos e herança.



Lição 70 – Classes e Objetos

A programação orientada a objetos (POO) é um paradigma de desenvolvimento em que o código de um aplicativo, pacote ou programa é estruturado em um conjunto objetos, cada qual com seus **atributos** (ou propriedades) e **métodos** específicos. Os atributos armazenam as características dos objetos, enquanto os métodos correspondem às ações que estes podem executar. Por exemplo, em um aplicativo de *streaming* de música como o Spotify, alguns dos objetos existentes seriam Usuário, Música e Playlist. Neste aplicativo, um Usuário poderia ser representado por seu “nome”, “idade”, “e-mail” e “cidade” – isto é, estas são as suas propriedades. O Usuário poderia executar ações como “ouvir uma Música”, “criar uma Playlist”, “inserir uma Música em uma Playlist”, “apagar uma Playlist”. Estes são os seus métodos.

Através da POO, podemos modelar as coisas que existem no mundo real de uma forma natural, estruturando-as em objetos. Como consequência, também fica mais fácil modelar as relações entre essas coisas. Considerando novamente o aplicativo de *streaming* de música, dois exemplos de relações entre objetos são “Usuário cria Playlist” e “Usuário ouve Música”.

No paradigma da POO, cada objeto é a instância de uma **classe**. As classes podem ser entendidas *templates* onde definimos os atributos e os métodos de um tipo específico de objeto. Para criar uma classe simples em Python, utilizamos a palavra reservada **class** e depois colocamos o código contendo a definição de seus atributos e métodos:

```
class Musica:
    especificação
```

Porém, antes de avançarmos e começarmos a desenvolver nossos primeiros programinhas orientados a objeto, é importante deixar claro que **classe e objeto não são a mesma coisa!** Conforme mencionado no parágrafo anterior, a classe é um *template*, ou seja, ela representa uma **estrutura**, mas não o conteúdo real. Por exemplo, poderíamos criar a classe Musica() e definir que a mesma armazenará três atributos: “nome”, “artista” e “estilos”. Tendo sido definida a classe Musica, aí sim poderemos instanciar diferentes objetos desta classe. Cada objeto é como se fosse uma variável do tipo Musica(), que conterá dados de uma única canção (como nome=“Wave”, artista=“Tom Jobim”, estilos=[“Bossa Nova”, “Jazz”]).



Lição 71 – Definindo Classes

Programa 114 – Definindo uma Classe e seus Atributos. O programa a seguir mostra a definição da classe Musica, com seus três atributos: “nome” (string), “artista” (string) e “estilos” (lista). A classe possui ainda um método, chamado “tocar()”. Vamos usar a nossa imaginação e fingir que este método faz a música ser tocada (na verdade, vamos apenas retornar uma mensagem dizendo que a música está em execução).

```
#P114: Definição de classe + método __init__()
class Musica:

    #construtor: inicializa as propriedades
    def __init__(self, p_nome, p_artista, p_estilos):
        self.nome = p_nome
        self.artista = p_artista
        self.estilos = p_estilos

    #método tocar()
    def tocar(self):
        return "tocando '{}' por {}...".format(self.nome, self.artista)
```

Saída [114]:

Se você executar esse programa, nada será exibido na saída. Por que? A resposta é simples: isto ocorre porque o programa contém apenas a definição de uma classe, neste caso a classe *Musica*. E sabemos que a classe é apenas um *template*. Nós ainda não estamos instanciando nenhum objeto desta classe.

O método `__init__()` (com dois sublinhados de cada lado) é um método reservado para uso pela própria linguagem Python. O `__init__()` é sempre **chamado automaticamente** quando instancia-se um objeto da classe (ou seja, você nunca precisará chamá-lo explicitamente). Na prática, ele é quase sempre empregado para atribuir valores para as propriedades no momento em que a classe for instanciada, como estamos fazendo em nosso programa. Por isso, o método é chamado de método **construtor** da classe.

Você deve ter observado que em nosso exemplo o método `__init__()` possui três argumentos (“p_nome”, “p_artista” e “p_estilos”) além do argumento **self**. Esse tal argumento **self** não possui esse nome por acaso. Na realidade, isso ocorre porque ele representa a própria instância do objeto, neste caso **uma música específica**. Quando fazemos `self.nome = p_nome` dentro do construtor, o que estamos dizendo para o Python é o seguinte: ao instanciar o objeto, faça o valor do atributo “nome” deste objeto (“self.nome”) ser igual ao valor do parâmetro “p_nome”. Analogamente iniciamos também os valores dos atributos “artista” (`self.artista = p_artista`) e “estilos” (`self.estilos = p_estilos`). Dessa forma, conseguimos garantir que toda instância de uma música tenha valores para os três atributos.

A nossa definição de classe termina com a especificação de um método chamado “tocar()”. Como já foi dito na lição anterior, os métodos são definidos dentro de uma classe e servem para executar algum tipo de ação. No caso do método “tocar()”, vamos imaginar que ele serviria para executar a música (na verdade ele irá somente retornar uma mensagem dizendo “tocando *a música tal* por *fulano*”). Veja que, assim como ocorre com o método `__init__()`, o primeiro argumento é sempre **self**. Para finalizar, um comentário importante: há outras formas para declarar classes, atributos e métodos. A que acabamos de apresentar pode ser considerada a mais simples.



Lição 72 – Instanciando Objetos

Talvez você ainda esteja um pouco confuso sobre como utilizar uma classe em um programa. Então calma, não precisa se preocupar! Nessa lição mostramos como instanciar objetos de uma classe, isto é, como criar diferentes objetos de uma dada classe.

Programa 115 – Instanciando Objetos. O programa a seguir mostra como instanciar dois objetos da classe Musica, passando as propriedades da música através do construtor. Também mostra-se como chamar um método.

#P115: Instanciando Objetos

```
class Musica:

    #construtor: inicializa as propriedades
    def __init__(self, p_nome, p_artista, p_estilos):
        self.nome = p_nome
        self.artista = p_artista
        self.estilos = p_estilos

    #método tocar()
    def tocar(self):
        return "tocando '{}' por {}".format(self.nome, self.artista)

#instancia um objeto do tipo Musica
m1 = Musica("Wave", "Tom Jobim", ["Bossa Nova", "Jazz"]);

#instancia outro objeto do tipo Musica
m2 = Musica("You Know I'm No Good", "Amy Winehouse", ["Jazz", "Pop", "Soul"]);

#chama o método tocar() para cada objeto
print('* música 1: ');
print(m1.tocar())

print('\n* Agora a música 2: ');
print(m2.tocar())
```

Saída [115]:

* música 1:
tocando 'Wave' por Tom Jobim...

* Agora a música 2:
tocando 'You Know I'm No Good' por Amy Winehouse...

Inicialmente temos a definição da classe Musica, da mesma forma que já havíamos feito no exemplo da lição anterior. As novidades vêm logo em seguida. Primeiro nós criamos duas músicas, uma do Tom Jobim (“m1”) e outra da Amy Winehouse (“m2”). Cada música está associada a um objeto diferente, isto é, “m1” e “m2” representam duas diferentes instâncias de objetos da classe Musica. Veja que para instanciar um objeto de uma classe, você deve usar o nome da classe, seguido de parênteses e passar os argumentos que estão definidos no método `__init__()`. Por exemplo, o objeto “m1” foi instanciado da seguinte forma:

```
m1 = Musica("Wave", "Tom Jobim", ["Bossa Nova", "Jazz"]);
```

Como o método `__init__()` é disparado automaticamente sempre que criamos uma nova instância de uma classe, então conseguimos atribuir o nome, artista e estilos da música com o comando acima. Veja que não é preciso passar o argumento `self`. Quando instanciamos o objeto, o Python automaticamente determina o que é `self` (uma música nesse caso) e simplesmente o passa para o método `__init__()`.

Depois de instanciar os objetos, podemos executar os métodos que foram definidos em sua classe. No caso da classe Musica, existe apenas um método definido: “tocar()”. Veja que para executá-lo basta especificar o nome do objeto, um ponto e depois o nome do método: `m1.tocar()`.



Lição 73 – Consultando e Modificando Atributos

Uma vez que um objeto tenha sido instanciado, podemos consultar e modificar os seus atributos a qualquer momento. Veja o exemplo a seguir.

Programa 116 – Consultando e Modificando Atributos.

#P116: Modificando Atributos

```
class Musica:

    #construtor: inicializa as propriedades
    def __init__(self, p_nome, p_artista, p_estilos):
        self.nome = p_nome
        self.artista = p_artista
        self.estilos = p_estilos

    #método tocar()
    def tocar(self):
        return "tocando '{}' por {}".format(self.nome, self.artista)

#cria um objeto e imprime seus atributos
m3 = Musica("O Mundo é um Moinho", "Cartola", ["Samba"]);

print("* dados originais:")
print("musica: ", m3.nome);
print("artista: ", m3.artista);
print("gênero: ", m3.estilos);

#modifica o valor do atributo "estilos"
m3.estilos = ["MPB", "Samba"]
print("\n* dados após modificação:")
print("musica: ", m3.nome);
print("artista: ", m3.artista);
print("gêneros: ", m3.estilos);
```

Saída [116]:

```
* dados originais:
musica: O Mundo é um Moinho
artista: Cartola
gênero: ['Samba']
```

```
* dados após modificação:
musica: O Mundo é um Moinho
artista: Cartola
gêneros: ['MPB', 'Samba']
```



Lição 74 – Atributos de Classe

Nos exemplos anteriores, mostramos como trabalhar com os atributos de instância, que são manipulados com o uso de `self` e são específicos de cada objeto. No entanto, também podemos definir atributos de classe, que sempre terão o mesmo valor para qualquer objeto instanciado da classe.

Programa 117 – Atributos de Classe. Neste exemplo, mostramos que todos os marcianos são verdes (atributo de classe), mas alguns são bons e outros são maus (atributo de objeto).

#P117: Atributos de Classe x Atributos de Instância

```
class Marciano:

    #atributo de classe - todo marciano é verde...
    cor = 'verde'

    #construtor: inicializa os atributos de instância
    def __init__(self, p_nome, p_tipo):
        self.nome = p_nome
        self.tipo = p_tipo

    #método get_info(): recupera o nome e a cor do Marciano
    def get_info(self):
        return "Me chamo {} e sou {}".format(self.nome, self.cor)

    #método get_mensagem(): retorna a frase que o Marciano dirá ao primeiro
    # ser humano que ele encontrar pela frente
    def get_mensagem(self, nome_ser_humano):
        if self.tipo.lower() == 'bom':
            return "{} , eu vim em missão de paz!".format(nome_ser_humano)
        else:
            return "{} , vou te abduzir e escravizar!".format(nome_ser_humano)

#cria dois marcianos, um bom e um mau
marciano_bom = Marciano("Tarkas", "bom");
marciano_mau = Marciano("Manhunter", "mau");

#o marciano bom encontrou a Anita
print(marciano_bom.get_info())
print(marciano_bom.get_mensagem("Ludmilla"))

#o marciano mau encontrou a Rihanna
print('-----')
print(marciano_mau.get_info())
print(marciano_mau.get_mensagem("Rihanna"))
```

Saída [117]:

Me chamo Tarkas e sou verde

Ludmilla, eu vim em missão de paz!

Me chamo Manhunter e sou verde

Rihanna, vou te abduzir e escravizar!

Independente da instância, todo marciano será ‘verde’, pois o atributo “cor” é um atributo de classe que possui ‘verde’ como valor fixo. Uma observação adicional: podemos utilizar tanto a sintaxe “self.cor” como “Marciano.cor” para recuperar a cor do Marciano. Mas para os atributos de instância, temos que utilizar sempre o “self.” (neste caso, “self.nome” e “self.tipo”). Faz sentido, não é?



Lição 75 – Herança

Herança é um processo pelo qual podemos definir uma nova classe (classe filha) tendo por base alguma outra classe (classe pai ou classe base). A nova classe herdará automaticamente todos os atributos e métodos da classe pai, mas poderá também definir seus próprios métodos e atributos adicionais. Além disso, a classe filha poderá **sobrescrever** (*override*) qualquer método ou atributo de classe da classe pai, o que significa modificar o comportamento original do método ou o valor original do atributo de classe. Todos esses conceitos são demonstrados no exemplo a seguir.

Programa 118 – Herança. Neste exemplo criamos uma classe pai chamada Animal e a partir dela, definimos duas classes filhas Cachorro e Gato. Depois instanciamos três objetos, dois do tipo Gato e um do tipo Cachorro.

```
#P118: Herança
```

```
#Classe pai
```

```
class Animal:
```

```
    #atributo de classe
```

```
    tipo = "Animal"
```

```
    #construtor: inicializa os atributos de instância
```

```
    def __init__(self, p_nome):
```

```
        self.nome = p_nome
```

```
    #método get_nome(): recupera o nome do animal
```

```
    def get_nome(self):
```

```
        return self.nome
```

```
#Classe filha - Cachorro
```

```
class Cachorro(Animal):
```

```
    #sobrescreve o atributo de classe
```

```
    tipo = "cão"
```

```
    #método conversar: como o animal conversa
```

```
    def conversar(self):
```

```
        return "au au au!"
```

```
#Classe filha - Gato
```

```
class Gato(Animal):
```

```
    #sobrescreve o atributo de classe
```

```
    tipo = "gato"
```

```

#método conversar: como o animal conversa
def conversar(self):
    return "miau miau miau!"

#cria três animais, dois gatos e um cachorro
nanquim = Gato("Nanquim");
pacato = Gato("Pacato");
dinamite = Cachorro("Dinamite");

#interação com os animais
print("{} é um {}".format(nanquim.get_nome(), nanquim.tipo))
print(nanquim.conversar())
print('-----')
print("{} é um {}".format(pacato.get_nome(), pacato.tipo))
print(pacato.conversar())
print('-----')
print("{} é um {}".format(dinamite.get_nome(), dinamite.tipo))
print(dinamite.conversar())

```

Saída [118]:

Nanquim é um gato
miau miau miau!

Pacato é um gato
miau miau miau!

Dinamite é um cão
au au au!

Segue a explicação. Animal é a classe pai, cuja definição engloba um atributo de classe (“tipo”, com valor ‘Animal’) e um atributo de instância (“nome”), cujo valor é atribuído através do método construtor.

A classe Gato é definida a partir de Animal (é filha de Animal), o que significa que ela herda automaticamente os atributos “tipo” e “nome” e o método construtor `__init__()`. No entanto, o atributo “tipo” é sobrescrito em Gato, o que significa que Gato anula o valor original de “tipo”, estabelecendo um novo valor (neste caso, ‘gato’). Além disso, Gato define um método adicional chamado “conversar()”, que não faz parte de Animal. Este método retorna o valor “miau miau miau”.

Por sua vez, a classe Cachorro também é definida a partir de Animal (é também filha de Animal), herdando suas propriedades e métodos e sobrescrevendo o atributo “tipo” (passa a valer ‘cão’). Cachorro também define um método chamado “conversar()”, que não faz parte de Animal e que retorna o valor “au au au”.

E assim encerramos esse capítulo que cobriu os conceitos elementares sobre programação orientada a objetos em Python. O conteúdo que apresentamos aqui representa apenas a “pontinha do iceberg” sobre POO, mas é o suficiente para que, a partir de agora, você se sinta mais confiante para estudar tópicos mais avançados, como encapsulamento, polimorfismo e classes abstratas, entre outros.

Capítulo X. Manipulação de Dados do Tipo Data/Hora

Um bom número de aplicações de ciência de dados requer a manipulação de informações do tipo data/hora. Um exemplo clássico é a análise de séries temporais, onde precisamos examinar uma série de dados coletados em incrementos sucessivos de tempo. Quando comparados com números e strings, os dados do tipo data/hora podem ser considerados um pouco mais complicados, pois além de possuírem formatação especial (as chamadas *timestrings* formatadas, como '%d-%m-%Y') eles são governados por uma série de regras para determinar os cálculos e resultados válidos (por exemplo, muitos cálculos devem levar em conta dias úteis, anos bissextos, etc.).

Felizmente, a linguagem Python oferece o módulo 'datetime', que faz parte da *standard library* e fornece os recursos necessários para facilitar a manipulação de informações sobre datas e horas. O objetivo deste capítulo é ensinar o leitor a utilizar as classes, métodos e *timestrings* oferecidas por este módulo.



Lição 76 – Módulo ‘datetime’

Quando estamos criando um programa para resolver um determinado problema (seja ele relacionado ou não à ciência de dados), muitas vezes nos deparamos com situações em que se torna preciso armazenar **datas de eventos** para representar certas informações. Veja alguns exemplos:

- Data de lançamento de um filme.
- Data inicial e data final de um projeto.
- Data de entrega de um pedido.
- Data de nascimento de um atleta.

Em outras situações, é necessário armazenar não apenas a data, mas também o **horário** ou **momento exato** da ocorrência de um evento:

- Horário da sessão de um filme.
- Horários de partida e de chegada de um voo.
- Momento (hora/minutos/segundos) em que um pedido de compra foi aprovado pela administradora do cartão de crédito.
- Tempo (segundos/milissegundos) em que um nadador completou a prova dos 100m livres.

Em situações como esta, o módulo ‘datetime’ deve ser utilizado para facilitar o processamento das informações. Nas próximas lições, mostraremos a receita básica para utilizar quatro classes disponibilizadas por este módulo: `date`, `datetime`, `time` e `timedelta`. O módulo oferece ainda classes para lidar com *time zones* (fuso horários), mas estas não serão cobertas em nosso livro.



Lição 77 – Operações sobre Datas

Vamos começar pela classe `date` que nos permite manipular datas (dia, mês e ano, sem a parte da hora).

Programa 119 – Operações Básicas sobre Datas. Este exemplo mostra como importar a classe `date` para depois criar e processar variáveis deste tipo.

```
#P119: Operações básicas sobre datas (date)
import datetime

#1-Cria uma data com o método date(). Basta passar ano, mes, dia
natal = datetime.date(2020,12,25)

#2-imprime a data e partes da mesma
print('tipo da variável:', type(natal)) #<class 'datetime.date'>
print('natal (data completa)=', natal)   #2020-12-25
print('natal - dia=', natal.day)         #25
```

```

print('natal - mês=', natal.month)           #12
print('natal - ano=', natal.year)            #2020
print('-----')

#Qual o dia da semana?
#0-Monday; 1-Tuesday; 2-Wednesday; 3-Thursday; 4-Friday; 5-Saturday; 6-Sunday
print('natal - dia da semana=', natal.weekday()) #4

#3-Soma e subtrai dias
dias = datetime.timedelta(10) #10 dias
d1 = natal - dias #resulta em 2020-12-15
d2 = natal + dias #resulta em 2021-01-04
x = d2 - d1 #resulta em 20 (diferença em dias entre as duas datas)

print('-----')
print('10 dias antes do natal=', d1) #2020-12-15
print('10 dias depois do natal=', d2) #2021-01-04
print('-----')
print('x=d2-d1=', x) #20 days, 0:00:00
print('type(x)=', type(x)) #class <'datetime.timedelta'>

#4-Comparando datas. Basta utilizar os operadores relacionais normalmente
print('-----')
print(d1 < d2) #True
print(d1 == d2) #False
print(d1 > d2) #False

```

Saída [119]:

```

tipo da variável: <class 'datetime.date'>
natal (data completa)= 2020-12-25
natal - dia= 25
natal - mês= 12
natal - ano= 2020
-----
natal - dia da semana= 4
-----
10 dias antes do natal= 2020-12-15
10 dias depois do natal= 2021-01-04
-----
x=d2-d1= 20 days, 0:00:00
type(x)= <class 'datetime.timedelta'>
-----
True
False
False

```

O programa está dividido em quatro partes.

- Na primeira, mostramos como utilizar o método `date()` para criar uma data (objeto da classe `datetime.date`), bastando para tal passar o ano, o mês e o dia como parâmetros, nesta ordem.

- Na segunda parte, imprimimos a data e também mostramos como utilizar as propriedades `day`, `month` e `year` para obter partes da mesma. Na Lição 80 veremos que também é possível obter os mesmos resultados utilizando um interessante método chamado `strftime()`. Ainda na segunda parte, apresentamos o método `weekday()`, que retorna um número que indica o dia da semana em que cai uma determinada data.
- A parte 3 mostra a receita que devemos utilizar para somar ou subtrair dias de uma determinada data (no caso, 10 dias). Basicamente, primeiro chamamos `datetime.timedelta(10)` para armazenar a duração em dias em uma variável da classe `timedelta`. Isso é necessário, pois o Python não consegue somar ou subtrair valores inteiros diretamente de uma data: ele só consegue operar com `timedeltas` (falaremos mais sobre o assunto ainda neste capítulo). Depois disso, tudo fica bem simples, pois basta utilizar o `timedelta` na subtração ou soma. O resultado de cada operação será uma nova variável do tipo `date`.
- Na parte 4, verificamos que a comparação de datas pode ser realizada de maneira trivial com o uso de operadores relacionais. Nas comparações, uma data mais antiga é sempre considerada menor do que uma mais recente.



Lição 78 – Operações sobre DateTimes

Uma variável da classe `datetime` nos permite manipular datas e horas conjuntamente (dia, mês, ano, hora, minuto, segundo, microssegundo e *time zone*). Os exemplos a seguir exploram as principais propriedades e métodos deste tipo de dado.

Programa 120 – Obtendo a Data/Hora do Sistema. Este exemplo mostra como importar a classe `datetime` e então recuperar a data/hora do sistema com o método `now()`.

```
#P120: Recuperando a data/hora do sistema
import datetime

agora = datetime.datetime.now()
print("tipo da variável:", type(agora))
print("data/hora corrente:", agora)
```

Saída [120]:

```
tipo da variável: <class 'datetime.datetime'>
data/hora corrente: 2020-02-22 15:45:16.685516
```

Neste exemplo, o programa foi executado no dia 22/02/2020 às 15:45:16.685516 (microssegundos).

Programa 121 – Operações Básicas sobre Datetimes. Este exemplo mostra como criar, modificar e extrair partes de variáveis do tipo `datetime`.

```
#P121: Operações básicas sobre datetimes
```

```

import datetime

#1.1-Podemos criar datetimes com o método datetime()...
d0 = datetime.datetime(2020,2,22,12,5,48)

#1.2-E também com strptime(), passando uma timestring + máscara de formatação
str1='2020-02-22'
str2='2020-02-22 12:05:48'

d1 = datetime.datetime.strptime(str1,"%Y-%m-%d")
d2 = datetime.datetime.strptime(str2,"%Y-%m-%d %H:%M:%S")

d3 = d2 + datetime.timedelta(365) #soma 365 dias à d2

#2-Imprime os datetimes
print(d0)    #2020-02-22 12:05:48
print(d1)    #2020-02-22 00:00:00
print(d2)    #2020-02-22 12:05:48
print(d3)    #2021-02-21 12:05:48

```

Saída [121]:

```

2020-02-22 12:05:48
2020-02-22 00:00:00
2020-02-22 12:05:48
2021-02-21 12:05:48

```

Segue a explicação do programa:

- Na parte 1, apresentamos duas maneiras para criar uma variável do tipo `datetime`:
 - A primeira é utilizando o método `datetime` passando os seguintes parâmetros (nesta ordem): ano, mês, dia (parâmetros obrigatórios), hora, minuto, segundo, microssegundo e *time zone* (parâmetros opcionais). Veja que este método é bem parecido com o que utilizamos no exemplo sobre a criação de variáveis `date`.
 - A segunda é armazenando a data/hora como um *timestring* para depois utilizar a função `strptime()`. Um *timestring* é nada mais do que uma notação alfanumérica “amigável” para expressar uma data ou data/hora. A notação deverá seguir alguma regra de formatação (ou máscara). Por exemplo, “25-12-2020” é uma *timestring* que casa com a máscara “%d-%m-%Y” (dia com dois dígitos + tracinho + mês com dois dígitos + tracinho + ano com quatro dígitos). Por sua vez, “2020-02-22 12:05:48”, exemplo mostrado em nosso programa, casa com a máscara “%Y-%m-%d %H:%M:%S”. Os códigos mais utilizados para a montagem de regras de formatação são relacionadas no Quadro 35:
- Na parte 2, apenas imprimimos o conteúdo de nossas variáveis `datetime`. Duas observações importantes:
 - A primeira é que a hora/minuto/segundo de “d1” foi automaticamente atribuída como ‘00:00:00’ porque essa variável é da classe `datetime`, mas foi criada apenas com a especificação do dia/mês/ano.

- A segunda é que podemos somar e subtrair dias à variáveis da classe `datetime` da mesma forma como fazemos com variáveis da classe `date`. Também podemos comparar dois `datetimes` da mesma forma que fazemos com objetos da classe `date` (para ver quem é maior ou menor).

Quadro 35 – códigos de formatação para timestrings

- **%d**: dia do ano, 01-31
- **%b**: nome do mês em Inglês, formato curto (ex: Dec);
- **%B**: nome do mês em Inglês, formato completo (ex: December);
- **%m**: mês como um número, 01-12;
- **%y**: ano com dois dígitos (ex: 98);
- **%Y**: ano com quatro dígitos (ex: 1998);
- **%H**: hora, 00-23;
- **%M**: minuto 00-59;
- **%S**: segundo 00-59;
- **%f**: microssegundo 000000-999999;

Um objeto do tipo `time` representa um horário (hora, minuto, segundo, microssegundo), sem armazenar nenhuma informação sobre o dia.

Programa 122 – Objeto `time`. Este exemplo mostra como criar um objeto da classe `time` utilizando o métodos `time()`.

```
#P122: trabalhando com a classe time
import datetime

t1=datetime.time(12,5,48)

print("tipo da variável:", type(t1))
print("valor de t1:", t1)           #12:05:48
```

Saída [122]:

```
tipo da variável: <class 'datetime.time'>
valor de t1: 12:05:48
```



Lição 80 – `strftime()`

Programa 123 – Método `strftime()`. Esse método permite extrair partes de um `date`, `datetime` ou `time` através da utilização de máscaras de formatação

```
#P123: método strftime
import datetime
```

```
#1-pegar a datetime do sistema
d = datetime.datetime.now()

#2-extraí partes com strftime
print("data/hora completa = ",d)
print("dia/mês/ano = ", d.strftime("%d/%m/%Y"))
print("hora:minuto:segundo = ", d.strftime("%H:%M:%S"))
print("só a hora = ", d.strftime("%H"))
print("só o ano com 2 dígitos = ", d.strftime("%y"))
print("só o nome abreviado do mês = ", d.strftime("%b"))
```

Saída [123]:

```
data/hora completa = 2020-02-22 16:15:50.652955
dia/mês/ano = 22/02/2020
hora:minuto:segundo = 16:15:50
só a hora = 16
só o ano com 2 dígitos = 20
só o nome abreviado do mês = Feb
```



Lição 81 – Classe timedelta

Um objeto `timedelta` representa a quantidade de tempo decorrido entre duas datas. No Python, podemos utilizar este objeto para medir intervalos de tempo ou para adicionar e subtrair intervalos de tempo a/de uma data.

Programa 124 – Classe `timedelta`. Este exemplo mostra formas diferentes para criar e utilizar objetos `timedelta`.

```
#P124: trabalhando com a classe timedelta
import datetime

#1-Cria os timedeltas
semana=datetime.timedelta(weeks=1) #7 dias
ano=datetime.timedelta(365) #365 dias
dia_e_meio=datetime.timedelta(days=1, hours=12) #1,5 dias

#2-imprime-os
print("semana=", semana) #7 days, 0:00:00
print("ano=", ano) #365 days, 0:00:00
print("dia_e_meio=", dia_e_meio) #1 day, 12:00:00

#3-soma cada timedelta ao datetime 01/01/2020 20:00:00
d = datetime.datetime(2020,1,1,20,0,0)
print('-----')
print("d=", d) #2020-01-01 20:00:00
print("d + semana=", d + semana) #2020-01-08 20:00:00
print("d + ano =", d + ano) #2020-12-31 20:00:00
print("d + dia_e_meio =", d + dia_e_meio) #2020-01-03 08:00:00
```

Saída [124]:

```
semana= 7 days, 0:00:00
ano= 365 days, 0:00:00
dia_e_meio= 1 day, 12:00:00
-----
d= 2020-01-01 20:00:00
d + semana= 2020-01-08 20:00:00
d + ano = 2020-12-31 20:00:00
d + dia_e_meio = 2020-01-03 08:00:00
```



Lição 82 – Utilizando o `from ... import`

Conforme vimos no Capítulo II, os módulos Python são arquivos “.py” contendo código escrito na linguagem. Ao longo do livro trabalhamos com diversos módulos da *standard library*, como `math`, `statistics`, `csv`, `re`, `json`, `sqlite` e, neste capítulo, o módulo `datetime`. Vimos ainda que em um programa Python, nós utilizamos o comando `import` para poder carregar um módulo:

```
import datetime
```

O detalhe que ainda não havíamos mencionado é que quando importamos um módulo da forma mostrada acima, o disponibilizamos em nosso programa atual como um **namespace** em separado. Um *namespace* é um sistema que o Python e outras linguagens utilizam para que seja possível haver um único nome para cada variável, tipo, função, classe, propriedade ou método. Na prática, isso é obtido através da notação que utiliza o ponto “.” separando o nome do módulo de algum de seus elementos. Veja o exemplo abaixo, onde estamos chamando o método `now()`, da classe `datetime`, do módulo `datetime`.

```
import datetime
agora = datetime.datetime.now()
```

Entretanto, você também pode importar fazendo uso do comando **`from ... import`**. Quando você faz a importação dessa maneira, torna-se possível referenciar os elementos de um módulo sem referenciar o nome do módulo. Veja o exemplo a seguir:

```
from datetime import datetime
agora = datetime.now()
```

Neste exemplo, o `import` carrega especificamente a classe `datetime`, do módulo `datetime`, em vez de carregar todo o módulo (por exemplo, não carregamos a classe `date` e nem a `timedelta`). Quando fazemos a importação dessa forma, não precisamos mais utilizar o nome do módulo prefixando o nome da classe (`datetime.datetime`); Ao contrário, podemos usar somente o nome da classe (`datetime`).

Em resumo: o uso da construção **`from ... import`** nos permite referenciar os elementos definidos de um módulo no *namespace* de nosso programa evitando a notação de pontos.

Anexo A - Temas Sugeridos para Estudo

Este livro introduziu os conceitos básicos sobre programação Python. O livro cobriu apenas os temas considerados relevantes para aqueles que desejam começar a trabalhar com Python para ciência de dados, iniciando pelos aspectos elementares da linguagem (variáveis, estruturas de dados, instruções de desvio e repetição e criação de funções), passando pelo processamento de strings e arquivos texto, prosseguindo com uma introdução à linguagem SQL e aos pacotes 'NumPy', 'pandas' e 'Matplotlib' e terminando com noções sobre orientação a objeto e manipulação de informações do tipo data/hora.

Por se tratar de um livro introdutório, **vários assuntos importantes não puderam ser abordados**. A seguir, relacionamos uma relação de temas sugeridos para aqueles que pretendem prosseguir em seus estudos com o objetivo de tornarem-se *pythonistas* avançados:

- **Python padrão**: este livro deixou de abordar alguns tópicos considerados um pouco mais avançados do Python padrão. Exemplos: **tratamento de exceções**, **aspectos avançados sobre orientação a objetos** (herança, polimorfismo, etc.), **programas em rede**, **programação paralela**, **programação funcional** com as funções `map()`, `filter()` e `reduce()`, processamento de **arquivos XML**, entre outros.
- **Configuração do ambiente Python**: o gerenciamento de pacotes e a criação de ambientes virtuais representam dois temas avançados, porém importantes para quem quer utilizar o Python de forma profissional.
- **pandas e scikit-learn**: no capítulo VII, apresentamos uma visão geral da biblioteca 'pandas', focando principalmente na importação de diferentes tipos de arquivo e na execução de operações básicas. No entanto, essa biblioteca é muito sofisticada e oferece um enorme conjunto de ferramentas para, seleção, limpeza e transformação de dados. Sendo assim, consideramos importante que você estude e aprenda mais sobre a 'pandas' para trabalhar em projetos profissionais de ciência de dados. Por sua vez, a 'scikit-learn' é a biblioteca que contém a implementação de diversos algoritmos para mineração de dados e aprendizado de máquina (algoritmos para classificação, análise de agrupamentos, regressão, seleção de atributos, etc.), representando uma ótima opção para os que finalizarem o estudo sobre a 'pandas'.
- **Jupyter Notebook** : trata-se de uma aplicação Web voltada para a criação de *notebooks*, que, basicamente, consistem em projetos de ciência de dados elaborados com o uso de Python, R ou outras tecnologias. O Jupyter Notebook é similar a uma IDE de programação, mas é muito melhor! Isto porque, um *notebook* pode armazenar tanto o código-fonte (programa propriamente dito), como as bases de dados utilizadas e os resultados obtidos em determinado processo de análise de dados (gráficos, tabelas, relatórios, modelos, etc.). Melhor ainda, os *notebooks* podem ser publicados na Web e compartilhados com outros usuários de uma maneira simples e rápida. Por estas razões, o Jupyter Notebook se tornou um dos mais populares aplicativos para ciência de dados.
- **scipy**: biblioteca que oferece rotinas numéricas eficientes para integração e otimização.

- **Keras, FastAI, TensorFlow e PyTorch:** estas bibliotecas são especialmente recomendadas para aqueles que desejam trabalhar com redes neurais e *deep learning*.
- **NLTK (*Natural Language Toolkit*):** conjunto de bibliotecas para mineração de texto e processamento de linguagem natural.
- **Web Scraping:** a linguagem Python padrão é dotada de excelentes recursos para aqueles que desejam trabalhar com Web scraping. Além das ótimas funções para tratamento de strings e do módulo 're' (expressões regulares), a própria *standard library* oferece o módulo 'socket' para conexão HTTP e recuperação de dados de páginas Web. E fora da *standard library*, existem ainda pacotes muito mais poderosos para Web scraping, como 'scrapy' e 'Beautiful Soup'.

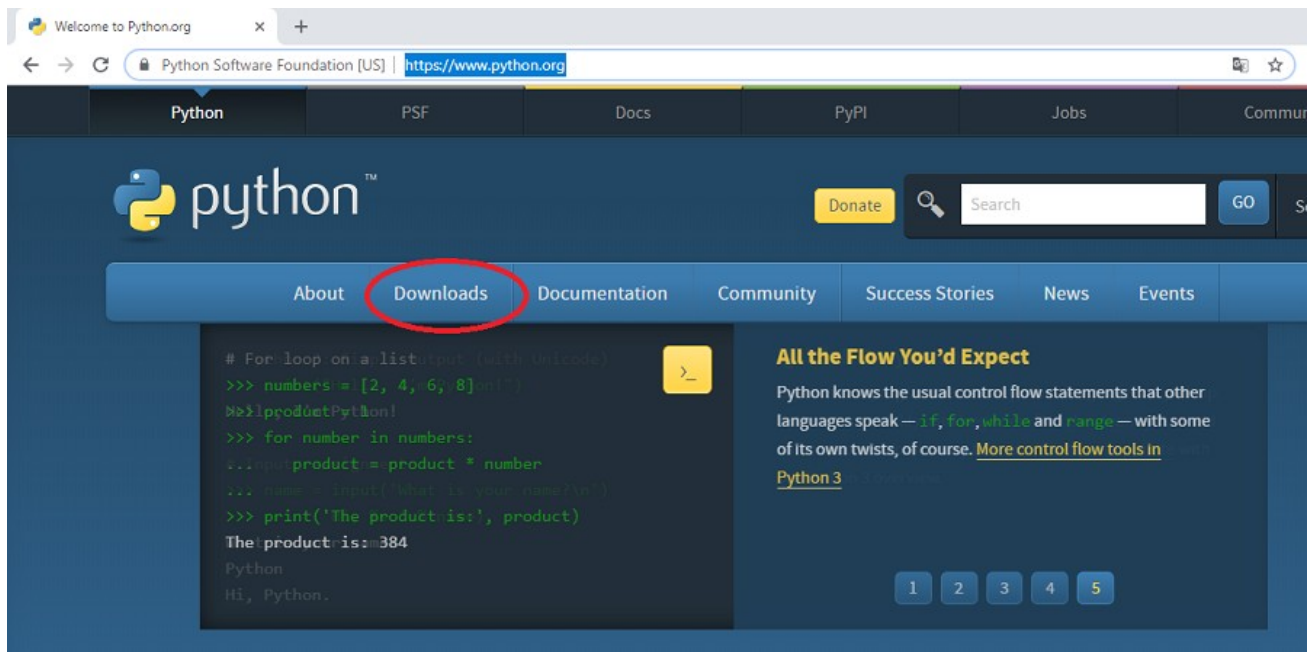
Anexo B - CPython

O texto a seguir apresenta os passos necessários para instalar e utilizar o ambiente Python padrão (conhecido como CPython), obtendo o instalador a partir do Website oficial do Python. Depois de instalar essa distribuição, você poderá acrescentar os pacotes de interesse separadamente utilizando o aplicativo "pip". A distribuição oficial é bem mais leve do que a distribuição WinPython e, além disso, está disponível para qualquer sistema operacional.

A.1- Instalação do Python

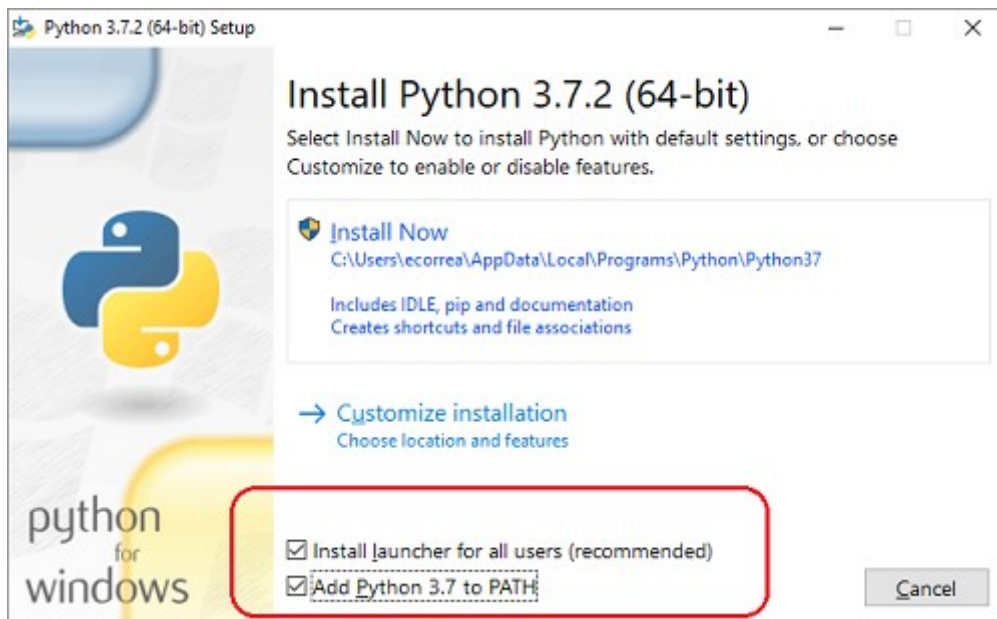
PASSO 1: DOWNLOAD

- Acesse o site <https://www.python.org/>, selecione Downloads (destacado na figura abaixo) e clique no botão referente à **versão 3** do Python que corresponda ao seu sistema operacional (Windows, Linux ou Mac).

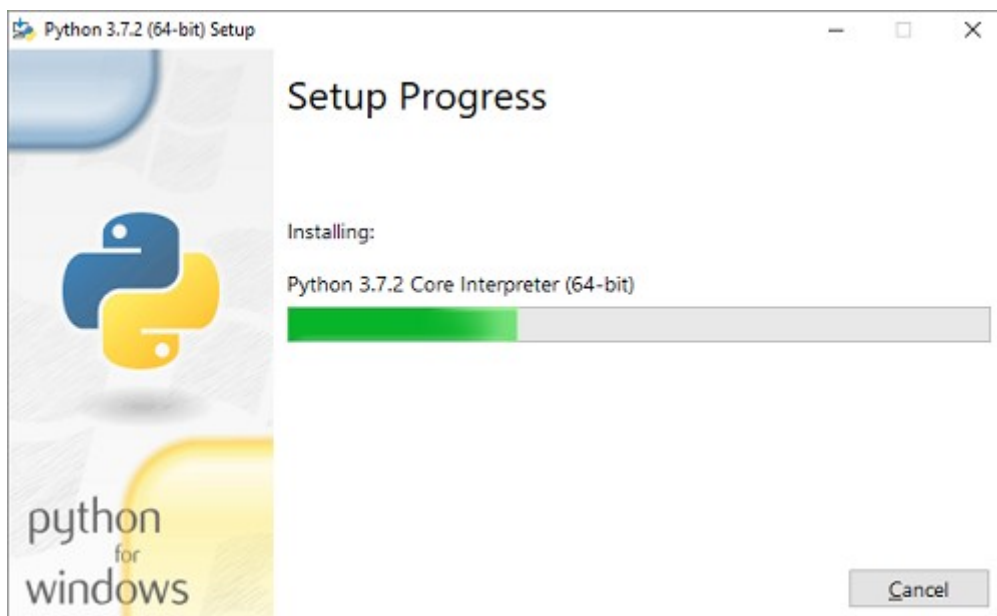


PASSO 2: INSTALAÇÃO

- Para realizar a instalação, basta executar o arquivo baixado. * * * **Importante!!!** se o seu sistema operacional for **Windows**, é interessante que você **marque as duas opções mostradas** figura da página a seguir (a opção "Add Python 3.7 to PATH" vem desmarcada, mas o ideal é que você a marque).



- A razão para a segunda opção não estar pré-selecionada por default é o fato de que muitas pessoas podem estar realizando a instalação com uma conta que não possui direito administrativo na máquina. Se você não marcá-la, o Python não será adicionado ao PATH do Windows e isto vai acabar atrapalhando a futura instalação de pacotes e a execução de programas através do prompt de comandos.
- De resto, é só prosseguir normalmente com a instalação clicando nos botões "Next" e aguardar até o processo ser finalizado.

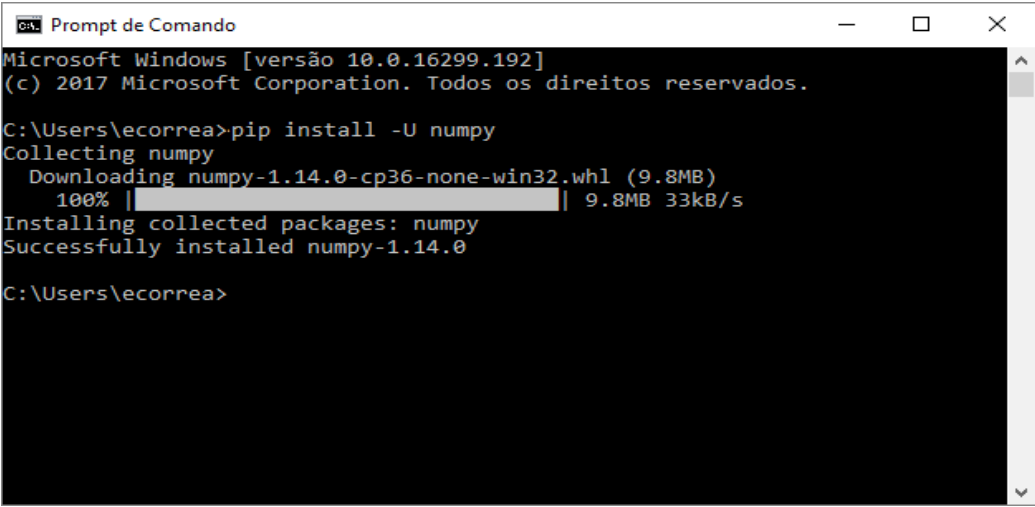


A.2- Instalação de Pacotes com o "pip"

- A forma mais utilizada para instalar pacotes é através do utilitário “pip”. Para usar este utilitário, é preciso abrir uma janela de terminal (no caso do Windows, "prompt de comandos") e digitar:

```
pip install -U nome_do_pacote
```

- O pacote especificado será então baixado e instalado. Caso ele já esteja instalado, será atualizado por uma versão mais nova (se existir). O exemplo a seguir, mostra a instalação do pacote 'Numpy':



```
cmd Prompt de Comando
Microsoft Windows [versão 10.0.16299.192]
(c) 2017 Microsoft Corporation. Todos os direitos reservados.

C:\Users\ecorrea>pip install -U numpy
Collecting numpy
  Downloading numpy-1.14.0-cp36-none-win32.whl (9.8MB)
    100% |#####| 9.8MB 33kB/s
Installing collected packages: numpy
Successfully installed numpy-1.14.0

C:\Users\ecorrea>
```

- Sendo assim, para instalar todos os pacotes que não fazem parte do Python padrão e que foram apresentados neste livro, você deve utilizar:

```
pip install -U numpy
pip install -U scipy
pip install -U matplotlib
pip install -U pandas
```

A.3- Instalação e Execução da IDE Spyder

- A seguir apresenta-se a forma de instalar e executar a IDE Spyder. A instalação também pode ser feita através do “pip” (obs.: essa instalação é bem mais demorada do que a instalação dos pacotes):

```
pip install -U spyder
```

- Após a instalação, você poderá executar o aplicativo digitando “spyder3” a partir da janela de terminal:

```
C:\Users\ecorreia>spyder3
C:\Users\ecorreia>_
```

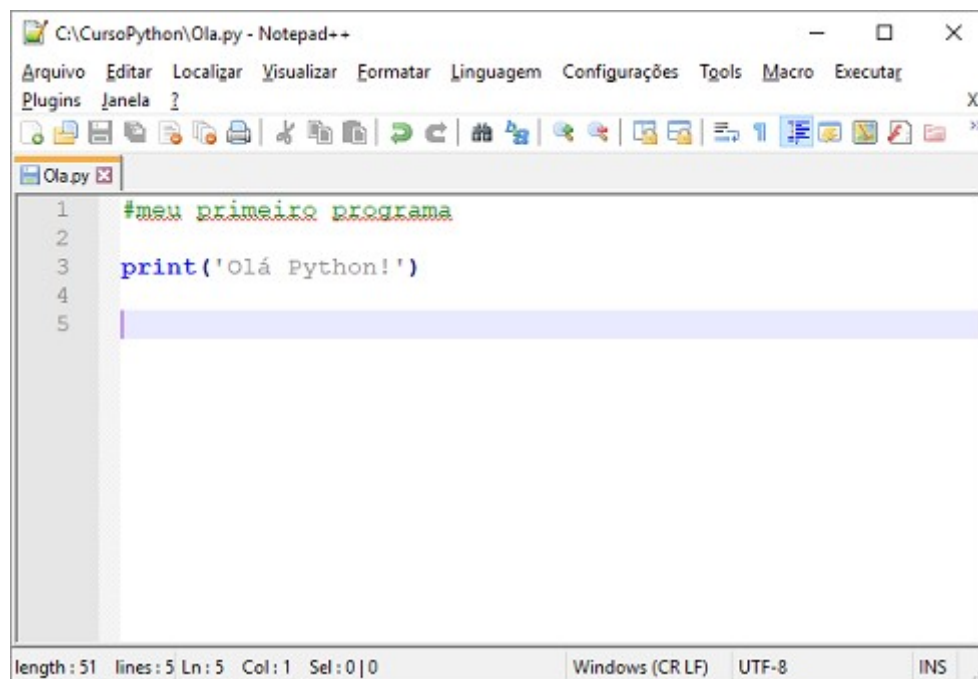
- Após alguns segundos, o aplicativo será carregado.



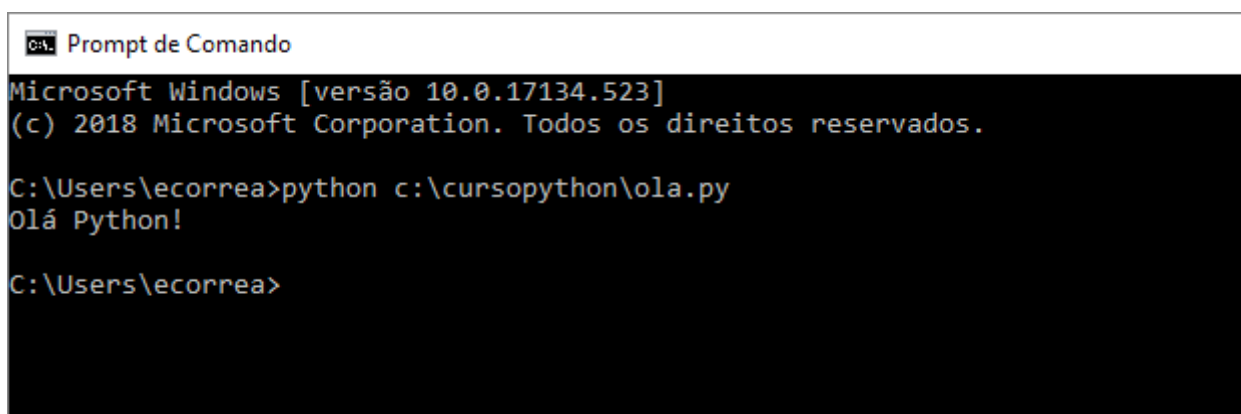
A.4- Criando e Executando Programas

O texto a seguir, mostra como criar, salvar e executar programas Python sem precisar usar a Spyder ou o IPython.

- Abra um editor de textos qualquer, como o Notepad++, e digite o programa. Ao terminar, salve-o com algum nome e a extensão “.py”. No exemplo da página a seguir, o programa foi salvo na pasta "C:\CursoPython" com o nome “Ola.py”.



- Após salvar, abra a janela de terminal ou prompt de comando. Daí, basta digitar:
`python nome_do_programa.py`
- Em nosso exemplo, como o programa está gravado na pasta "C:\CursoPython", foi preciso especificar o caminho completo: `python c:\cursopython\ola.py` (como mostra a figura abaixo).



Bibliografia

CORRÊA, E. Pandas Python: Data Wrangling para Ciência de Dados. São Paulo: Casa do Código, 2020.

DATA CAMP. Python tutorials. DataCamp, 2019. Disponível em: <<https://www.datacamp.com/community/tutorials?tag=python>>. Acesso em: 06 fev. 2019.

DATAQUEST. Python datetime tutorial: Manipulate times, dates, and time spans. DataQuest, 2019. Disponível em: <<https://www.dataquest.io/blog/python-datetime-tutorial/>>. Acesso em: 23 fev. 2020.

DOWNEY, A. B. Think Stats: exploratory data analysis in Python. Needham: Green Tea Press, version 2.0.38, 2014.

HALE, J. The Most in Demand Skills for Data Scientists. KDnuggets, 2018. Disponível em: <<https://www.kdnuggets.com/2018/11/most-demand-skills-data-scientists.html>>. Acesso em: 30 dez. 2018.

HAND, D. Data Science. Wiley StatsRef-Statistics, Reference Online (to appear). 2018.

HELLMANN, D. The Python 3 Standard Library by Example. Ann Arbor: Addison-Wesley Professional, 2011.

HELLMANN, D. Python Module of the Week. Disponível em: <<https://pymotw.com/3/>>. Acesso em: 15 jan. 2019.

HOODA, S. What is the Best Python IDE for Data Science?. KDnuggets, 2018. Disponível em: <<https://www.kdnuggets.com/2018/11/best-python-ide-data-science.html>>. Acesso em: 02 jan. 2018.

JARGAS, A. Expressões Regulares: uma abordagem divertida. Rio de Janeiro: Novatec, 5 ed., 2016.

KDNUGGETS. <<https://www.kdnuggets.com/>>. Acesso em: 06 fev. 2019.

MCKINNEY, W. Python for Data Analysis: data wrangling with pandas, numpy, and ipython. Sebastopol: O'Reilly, 2ed, 2018.

PETERS, T. The Zen of Python. Python Software Foundation, 2004. Disponível em: <<https://www.python.org/dev/peps/pep-0020/>>. Acesso em: 02 jan. 2019.

PIATETSKY, G. Python vs R – Who Is Really Ahead in Data Science, Machine Learning? KDnuggets, 2017. Disponível em: <<https://www.kdnuggets.com/2017/09/python-vs-r-data-science-machine-learning.html>>. Acesso em: 29 dez. 2018.

PIP 18.1. The PyPA recommended tool for installing Python packages. Python Software Foundation, 2019. Disponível em: <<https://pypi.org/project/pip/>>. Acesso em: 02 jan. 2019.

PYHTON. Python 3.7.2 documentation. Python Software Foundation, 2019. Disponível em: <<https://docs.python.org/3/>>. Acesso em: 02 jan. 2019.

PYHTON. Python success stories. Python Software Foundation, 2019. Disponível em: <<https://www.python.org/about/success/>>. Acesso em: 02 jan. 2019.

REALPYHTON. Object-oriented programming (OOP) in Python 3. Real Python, 2019. Disponível em: <<https://realpython.com/python3-object-oriented-programming/>>. Acesso em: 26 fev. 2020.

SEVERANCE, C. R. Python for Everybody: exploring data using python 3. Ann Arbor: Charles Severance, 2013.

THEUWISSEN, M. R vs Python for Data Science: The Winner is ... KDnuggets, 2015. Disponível em: <<https://www.kdnuggets.com/2015/05/r-vs-python-data-science.html>>. Acesso em: 29 dez. 2018.

WIKIBOOKS. Python programming. en.wikibooks.org, 2015. Disponível em: <https://en.wikibooks.org/wiki/Python_Programming>. Acesso em: 30 dez. 2018.

WINPYTHON. Python programming. WinPython, 2018. Disponível em: <<http://winpython.github.io/>>. Acesso em: 23 dez. 2018.

YEGULALP, S. Anaconda, Cpython, PyPy, and More: Know your Python Distributions. Infoworld, 2018. Disponível em: <<https://www.infoworld.com/article/3267976/python/anaconda-cpython-pypy-and-more-know-your-python-distributions.html>>. Acesso em: 23 dez. 2018.

Sobre o Autor

Eduardo Corrêa Gonçalves cursou Doutorado em Ciência da Computação pela UFF (2015) com período como pesquisador visitante na University of Kent, no Reino Unido. Também cursou Mestrado (2004) e Graduação (1999) em Ciência da Computação pela UFF. Atualmente, trabalha como analista de banco de dados no Instituto Brasileiro de Geografia e Estatística (IBGE) e também atua como professor colaborador na Escola Nacional de Ciências Estatísticas (ENCE-IBGE).

Currículo Lattes: <http://buscatextual.cnpq.br/buscatextual/visualizacv.do?id=K4137241P3>

