



Python do ZERO

à Programação Orientada a Objetos

Fernando Feltrin

edição atualizada e ampliada.

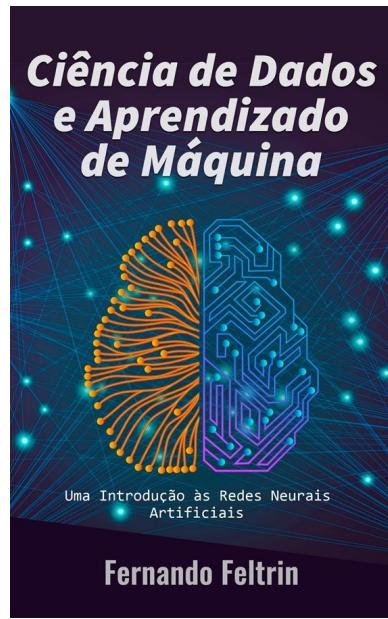
3 LIVROS

+ de 400 páginas



*Ciência de Dados
e Aprendizado
de Máquina*

Fernando Feltrin



LIVRO 1 – Python do ZERO à Programação Orientada a Objetos

LIVRO 2 – Programação Orientada a Objetos com Python

LIVRO 3 – Ciência de Dados e Aprendizado de Máquina

Fernando Feltrin

Índice

[Índice](#)

[Por quê programar? E por quê em Python?](#)

[Metodologia](#)

[1 – Introdução](#)

[Por quê Python?](#)

[Um pouco de história](#)

[Guido Van Rossum](#)

[A filosofia do Python](#)

[Empresas que usam Python](#)

[O futuro da linguagem](#)

[Python será o limite?](#)

[2 – Ambiente de programação](#)

[Linguagens de alto, baixo nível e de máquina](#)

[Ambientes de desenvolvimento integrado](#)

[Principais IDEs](#)

[3 – Lógica de programação](#)

[Algoritmos](#)

[Sintaxe em Python](#)

[Palavras reservadas](#)

[Análise léxica](#)

[Indentação](#)

[4 – Estrutura básica de um programa](#)

[5 – Tipos de dados](#)

[6 – Comentários](#)

[7 – Variáveis / objetos](#)

[Declarando uma variável](#)

[Declarando múltiplas variáveis](#)

[Declarando múltiplas variáveis \(de mesmo tipo\)](#)

[8 – Funções Básicas](#)

[Função print\(\)](#)

[Função input\(\)](#)

[Explorando a função print\(\)](#)

[Interação entre variáveis](#)

[Conversão de tipos de dados](#)

9 – Operadores

[Operadores de Atribuição](#)

[Atribuições especiais](#)

[Operadores aritméticos](#)

[Operadores Lógicos](#)

[Tabela verdade](#)

[Operadores de membro](#)

[Operadores relacionais](#)

[Operadores usando variáveis](#)

[Operadores usando condicionais](#)

[Operadores de identidade](#)

10 – Estruturas condicionais

[Ifs, elifs e elses](#)

[And e Or dentro de condicionais](#)

[Condicionais dentro de condicionais](#)

[Simulando switch/case](#)

11 - Estruturas de repetição

[While](#)

[For](#)

12 – Strings

[Trabalhando com strings](#)

[Formatando uma string](#)

[Convertendo uma string para minúsculo](#)

[Convertendo uma string para maiúsculo](#)

[Buscando dados dentro de uma string](#)

[Desmembrando uma string](#)

[Alterando a cor de um texto](#)

[Alterando a posição de exibição de um texto](#)

[Formatando a apresentação de números em uma string](#)

13 – Listas

[Adicionando dados manualmente](#)

[Removendo dados manualmente](#)

[Removendo dados via índice](#)

[Verificando a posição de um elemento](#)

[Verificando se um elemento consta na lista](#)

[Formatando dados de uma lista](#)

[Listas dentro de listas](#)

[Tuplas](#)

[Pilhas](#)

[Adicionando um elemento ao topo de pilha](#)

[Removendo um elemento do topo da pilha](#)

[Consultando o tamanho da pilha](#)

[14 – Dicionários](#)

[Consultando chaves/valores de um dicionário](#)

[Consultando as chaves de um dicionário](#)

[Consultando os valores de um dicionário](#)

[Mostrando todas chaves e valores de um dicionário](#)

[Manipulando dados de um dicionário](#)

[Adicionando novos dados a um dicionário](#)

[15 – Conjuntos numéricos](#)

[União de conjuntos](#)

[Interseção de conjuntos](#)

[Verificando se um conjunto pertence ao outro](#)

[Diferença entre conjuntos](#)

[16 – Interpolação](#)

[Avançando com interpolações](#)

[17 – Funções](#)

[Funções predefinidas](#)

[Funções personalizadas](#)

[Função simples, sem parâmetros](#)

[Função composta, com parâmetros](#)

[Função composta, com *args e **kwargs](#)

[18 – Comandos dir\(\) e help\(\)](#)

[19 – Builtins e bibliotecas pré-alocadas](#)

[Importando bibliotecas](#)

[20 – Módulos e pacotes](#)

[Modularização](#)

[21 – Programação orientada a objetos](#)

[Classes](#)

[Definindo uma classe](#)

[Alterando dados/valores de uma instância](#)

[Aplicando recursividade](#)

[Herança](#)

[Polimorfismo](#)

[Encapsulamento](#)

[22 – Tracebacks / exceções](#)

[Comandos try, except e finally](#)

[23 – Considerações finais](#)

[24 – Introdução / Livro 2 - Programação Orientada a Objetos com Python](#)

[25 - Ambiente de Programação](#)

[JetBrains PyCharm](#)

[Microsoft Visual Studio Code](#)

[Jupyter Notebook](#)

[26 - Programação Orientada a Objetos](#)

[Objetos e Classes](#)

[Variáveis vs Objetos](#)

[Criando uma classe vazia](#)

[Atributos de classe](#)

[Manipulando atributos de uma classe](#)

[Métodos de classe](#)

[Método construtor de uma classe](#)

[Escopo / Indentação de uma classe](#)

[Classe com parâmetros opcionais](#)

[Múltiplos métodos de classe](#)

[Interação entre métodos de classe](#)

[Estruturas condicionais em métodos de classe](#)

[Métodos de classe estáticos e dinâmicos](#)

[Getters e setters](#)

[Encapsulamento](#)

[Associação de classes](#)

[Agregação e composição de classes](#)

[Herança Simples](#)

[Cadeia de heranças](#)

[Herança Múltipla](#)

[Sobreposição de membros](#)

[Classes abstratas](#)

[Polimorfismo](#)

[Sobrecarga de operadores](#)

[Tratando exceções](#)

[Capítulo Final](#)

[27 - Introdução / Livro 3 – Ciência de Dados e Aprendizado de Máquina](#)

Nota do Autor

Afinal de contas, o que é Ciência de Dados e Aprendizado de Máquina?

Como se trabalha com Ciência de Dados? Quais ferramentas utilizaremos?

Quando e como utilizaremos tais ferramentas?

Qual a Abordagem que Será Utilizada?

28 – Preparação do Ambiente de Trabalho

Instalação do Python 3

Instalação da Suite Anaconda

Instalação das Bibliotecas e Módulos Adicionais

29 – Teoria e Prática em Ciência de Dados

Teoria Sobre Redes Neurais Artificiais

Aprendizado de Máquina

Perceptrons

Perceptron Multicamada

Deep Learning

Rotinas de uma Rede Neural Artificial

30 – Aprendizado de Máquina

Perceptron de Uma Camada – Modelo Simples

Perceptron de Uma Camada – Tabela AND

Perceptron Multicamada – Tabela XOR

31 – Redes Neurais Artificiais

Classificação Multiclasse – Aprendizado de Máquina Simples – Iris Dataset

Classificação Multiclasse via Rede Neural Artificial – Iris Dataset

Classificação Binária via Rede Neural Artificial – Breast Cancer Dataset

Regressão de Dados de Planilhas Excel – Autos Dataset

Regressão com Múltiplas Saídas – VGDB Dataset

Previsão Quantitativa – Publi Dataset

32 – Redes Neurais Artificiais Convolucionais

Reconhecimento de Caracteres – Digits Dataset

Classificação de Dígitos Manuscritos – MNIST Dataset

Classificação de Imagens de Animais – Bichos Dataset

Classificação a Partir de Imagens – TensorFlow – Fashion Dataset

33 – Redes Neurais Artificiais Recorrentes

Previsão de Séries Temporais – Bolsa Dataset

34 – Outros Modelos de Redes Neurais Artificiais

Mapas Auto Organizáveis – Kmeans – Vinhos Dataset

Sistemas de Recomendação – Boltzmann Machines

[35 – Parâmetros Adicionais](#)

[36 – Capítulo Resumo](#)

[37 – Considerações Finais](#)

Por quê programar? E por quê em Python?

No âmbito da tecnologia da informação, basicamente começamos a dividir seus nichos entre a parte física (hardware) e sua parte lógica (software), e dentro de cada uma delas existe uma infinidade de subdivisões, cada uma com suas particularidades e usabilidades diferentes.

O aspirante a profissional de T.I. pode escolher entre várias muitas áreas de atuação, e mesmo escolhendo um nicho bastante específico ainda assim há um mundo de conhecimento a ser explorado. Dentro da parte lógica um dos diferenciais é a área da programação, tanto pela sua complexidade quanto por sua vasta gama de possibilidades.

Sendo assim um dos diferenciais mais importantes do profissional de tecnologia moderno é o mesmo ter certa bagagem de conhecimento de programação. No âmbito acadêmico existem diversos cursos, que vão de análise e desenvolvimento de sistemas até engenharia da computação, e da maneira como esses cursos são organizados você irá reparar que sempre haverá em sua grade curricular uma carga horária dedicada a programação. No Brasil a linguagem de programação mais popularmente utilizada nos cursos de tecnologia é C ou uma de suas vertentes, isso se dá pelo fato de C ser uma linguagem ainda hoje bastante popular e que pode servir de base para tantas outras.

Quando estamos falando especificamente da área da programação existe uma infinidade de linguagens de programação que foram sendo desenvolvidas ao longo do tempo para suprir a necessidade de criação de softwares que atendessem uma determinada demanda. Poderíamos dedicar um capítulo inteiro mostrando apenas as principais e suas características, mas ao invés disso vamos nos focar logo em Python.

Hoje com a chamada internet das coisas, data science, machine learning, além é claro da criação de softwares, jogos e sistemas, mais do que nunca foi preciso profissionais da área que soubessem programação. Python é uma linguagem idealizada e criada na década de 80, mas que se mostra hoje uma das mais modernas e promissoras, devido sua facilidade de aprendizado e sua capacidade de se adaptar a qualquer situação. Se você buscar qualquer comparativo de Python em relação a outras linguagens de programação garanto que em 95% dos casos Python sairá em vantagem.

Python pode ser a sua linguagem de programação definitiva, ou abrir muitas portas para aprender outras mais, já que aqui não existe uma real concorrência, a melhor linguagem sempre será aquela que irá se adaptar melhor ao programador e ao projeto a ser desenvolvido.

Sendo assim, independentemente se você já é programador de outra linguagem ou se você está começando do zero, espero que o conteúdo deste pequeno livro seja de grande valia para seu aprendizado dentro dessa área incrível.

Metodologia

Este material foi elaborado com uma metodologia autodidata, de forma que cada conceito será explicado de forma progressiva, sucinta e exemplificado em seguida.

Cada tópico terá seus exemplos e devida explicação, assim como sempre que necessário você terá o código na íntegra e em seguida sua ‘engenharia reversa’, explicando ponto a ponto o que está sendo feito e os porquês de cada argumento dentro do código.

Cada tópico terá ao menos um exemplo; cada termo dedicado a programação terá destaque para que você o diferencie em meio ao texto ou explicações.

Desde já tenha em mente que aprender a programar requer atenção e mais do que isso, muita prática. Sendo assim, recomendo que sempre que possível pratique recriando os códigos de exemplo e não tenha medo que testar diferentes possibilidades em cima deles.

Dadas as considerações iniciais, mãos à obra!!!

1 – Introdução

Quando estamos iniciando nossa jornada de aprendizado de uma linguagem de programação é importante que tenhamos alguns conceitos bem claros já logo de início. O primeiro deles é que aprender uma linguagem de programação não é muito diferente do que aprender outro idioma falado, haverá uma sintaxe e uma sequência lógica de argumentos a se respeitar a fim de que seus comandos façam sentido e funcionem.

Com certeza quando você começou a aprender inglês na escola você começou pelo verbo **to be** e posteriormente foi incrementando com novos conceitos e vocabulário, até ter certo domínio sobre o mesmo. Em uma linguagem de programação não é muito diferente, há uma sintaxe, que é a maneira com que o interpretador irá reconhecer seus comandos, e há também uma lógica de programação a ser seguida uma vez que queremos através de linhas/blocos de código dar instruções ao computador e chegar a algum resultado.

Se você pesquisar, apenas por curiosidade, sobre as linguagens de programação você verá que outrora elas eram extremamente complexas, com uma curva de aprendizado longo e que por fim eram pouco eficientes ou de uso bastante restrito. As linguagens de alto nível, como Python, foram se modernizando de forma a que hoje é possível fazer muito com pouco, de forma descomplicada e com poucas linhas de código já estaremos criando programas e/ou fazendo a manutenção dos mesmos.

Se você realmente tem interesse por essa área arrisco dizer que você irá adorar Python e programação em geral. Então chega de enrolação e vamos começar... do começo.

Por quê Python?

Como já mencionei anteriormente, um dos grandes diferenciais do Python, que normalmente é o chamativo inicial por quem busca uma linguagem de programação, é sua facilidade, seu potencial de fazer mais com menos. Não desmerecendo outras linguagens, mas é fato que Python foi desenvolvida para ser descomplicada. E por quê fazer as coisas da forma mais difícil se existem ferramentas para torná-las mais fáceis?

Existe uma piada interna do pessoal que programa em Python que diz:

“- A vida é curta demais para programar em outra linguagem senão em Python. “

Como exemplo veja três situações, um simples programa que exibe em tela a mensagem “Olá Mundo!!!” escrito em C, em JAVA, e por fim em Python.

Olá Mundo!!! em C

```
#include<stdio.h>
int main (void)
{
printf("Olá Mundo!!!\n");
return 0;
}
```

Olá Mundo!!! em JAVA

```
public class hello {
public static void main (String arg[]){
    System.out.println("Olá Mundo!!!");
}
}
```

Olá Mundo!!! em Python

```
print('Olá Mundo!!!')
```

Em todos os exemplos acima o retorno será uma mensagem exibida em tela para o usuário dizendo: **Olá Mundo!!!** (*Garanto que agora a piada fez sentido...)

De forma geral o primeiro grande destaque do Python frente a outras linguagens é sua capacidade de fazer mais com menos, e em todas as situações que conheço este padrão se repetirá, será muito mais fácil, mais rápido, com menos linhas de código, programar em Python.

Outro grande diferencial do Python em relação a outras linguagens de programação é o fato dela ser uma linguagem interpretada, em outras palavras, o ambiente de programação que você irá trabalhar tem capacidade de rodar o código em tempo real e de forma nativa, diferente de outras linguagens que tem que estar emulando uma série de parâmetros do sistema ou até mesmo compilando o código para que aí sim seja possível testá-lo.

Outro diferencial é a sua simplicidade sintática, posteriormente você aprenderá sobre a sintaxe Python, mas por hora apenas imagine que em programação (outras linguagens) muitas vezes temos uma grande ideia e ao codificá-la acabamos nos frustrando porque o código simplesmente não funciona, e em boa parte das vezes é por conta de um ponto ou vírgula que ficou sobrando ou faltando no código. Python por ser uma linguagem interpretada deveria sofrer ainda mais com esses problemas, mas o que ocorre é ao contrário, o interpretador foca nos comandos e seus parâmetros, os pequenos erros de sintaxe não farão com que o código não funcione...

Por fim, outro grande diferencial do Python se comparado a outras linguagens de programação é que ela nativamente possui um núcleo capaz de trabalhar com uma quantidade enorme de dados de forma bastante tranquila, o que a fez virar a queridinha de quem trabalha com data science, machine learning, blockchain, e outras tecnologias que trabalham e processam volumes enormes de dados.

Na verdade, eu poderia escrever outro livro só comparando Python com outras linguagens e mostrando suas vantagens, mas acho que por hora já é o suficiente para lhe deixar empolgado, e ao longo desse curso você irá descobrir um potencial muito grande em Python.

Um pouco de história

Em 1989, através do Instituto de Pesquisa Nacional para Matemática e Ciência da Computação, Guido Van Rossum publicava a primeira versão do Python. Derivada do C, a construção do Python se deu inicialmente para ser uma alternativa mais simples e produtiva do que o próprio C. Por fim em 1991 a linguagem Python ganhava sua “versão estável” e já funcional, começando a gerar também uma comunidade dedicada a aprimorá-la. Somente em 1994 foi lançada sua versão 1.0, ou seja, sua primeira versão oficial e não mais de testes, e de lá para cá houveram gigantescas melhorias na linguagem em si, seja em estrutura quanto em bibliotecas e plugins criadas pela comunidade para implementar novos recursos a mesma e a tornar ainda mais robusta.

Atualmente o Python está integrado em praticamente todas novas tecnologias, assim como é muito fácil implementá-la em sistemas “obsoletos”. Grande parte das distribuições Linux possuem Python nativamente e seu reconhecimento já fez com que, por exemplo, virasse a linguagem padrão do curso de ciências da computação do MIT desde 2009.

Guido Van Rossum

Não posso deixar de falar, ao menos um pouco, sobre a mente por trás da criação do Python, este cara chamado Guido Van Rossum. Guido é um premiado matemático e programador que hoje se dedica ao seu emprego atual na Dropbox. Guido já trabalhou em grandes empresas no passado e tem um grande reconhecimento por estudantes e profissionais da computação.

Dentre outros projetos, Guido em 1991 lançou a primeira versão de sua própria linguagem de programação, o Python, que desde lá sofreu inúmeras melhorias tanto pelos seus desenvolvedores quanto pela comunidade e ainda hoje ele continua supervisionando o desenvolvimento da linguagem Python, tomando as decisões quando necessário.

A filosofia do Python

Python tem uma filosofia própria, ou seja, uma série de porquês que são responsáveis por Python ter sido criada e por não ser “só mais uma linguagem de programação”.

Por Tim Peters, um influente programador de Python

1. *Bonito é melhor que feio.*
2. *Explícito é melhor que implícito.*
3. *Simples é melhor que complexo.*
4. *Complexo é melhor que complicado.*
5. *Plano é melhor que aglomerado.*
6. *Escasso é melhor que denso.*
7. *O que conta é a legibilidade.*
8. *Casos especiais não são especiais o bastante para quebrar as regras.*
9. *A natureza prática derruba a teórica.*
10. *Erros nunca deveriam passar silenciosamente.*
11. *a menos que explicitamente silenciasse.*
12. *Diante da ambiguidade, recuse a tentação de adivinhar.*
13. *Deveria haver um -- e preferivelmente só um -- modo óbvio para fazer as coisas.*
14. *Embora aquele modo possa não ser óbvio a menos que você seja holandês.*
15. *Agora é melhor que nunca.*
16. *Embora nunca é frequentemente melhor que *agora mesmo*.*
17. *Se a implementação é difícil para explicar, isto é uma ideia ruim.*
18. *Se a implementação é fácil para explicar, pode ser uma ideia boa.*
19. *Namespaces são uma grande ideia -- façamos mais desses!*

Essa filosofia é o que fez com que se agregasse uma comunidade enorme disposta a investir seu tempo em Python. Em suma, programar em Python deve ser simples, de fácil aprendizado, com código de fácil leitura, enxuto, mas inteligível, capaz de se adaptar a qualquer necessidade.

Empresas que usam Python

Quando falamos de linguagens de programação, você já deve ter reparado que existem inúmeras delas, mas basicamente podemos dividi-las em duas grandes categorias: Linguagens específicas e/ou Linguagens Generalistas. Uma linguagem específica, como o próprio nome sugere, é aquela linguagem que foi projetada para atender a um determinado propósito fixo, como exemplo podemos citar o PHP e o HTML, que são linguagens específicas para web. Já linguagens generalistas são aquelas que tem sua aplicabilidade em todo e qualquer propósito, e nem por isso ser inferior às específicas.

No caso do Python, ela é uma linguagem generalista bastante moderna, é possível criar qualquer tipo de sistema para qualquer propósito e plataforma a partir dela. Só para citar alguns exemplos, em Python é possível programar para qualquer sistema operacional, web, mobile, data science, machine learning, blockchain, etc... coisa que outras linguagens de forma nativa não são suficientes ou práticas para o programador, necessitando uma série de gambiarras para realizar sua codificação, tornando o processo mais difícil.

Como exemplo de aplicações que usam parcial ou totalmente Python podemos citar YouTube, Google, Instagram, Dropbox, Quora, Pinterest, Spotify, Reddit, Blender 3D, BitTorrent, etc... Apenas como curiosidade, em computação gráfica dependendo sua aplicação uma engine pode trabalhar com volumosos dados de informação, processamento e renderização, a Light and Magic, empresa subsidiária da Disney, que produz de maneira absurda animações e filmes com muita computação gráfica, usa de motores gráficos escritos e processados em Python devido sua performance.

O futuro da linguagem

De acordo com as estatísticas de sites especializados, Python é uma das linguagens com maior crescimento em relação às demais no mesmo período, isto se deve pela popularização que a linguagem recebeu após justamente grandes empresas declarar que a adotaram e comunidades gigantescas se formarem para explorar seu potencial. Em países mais desenvolvidos tecnologicamente até mesmo escolas de ensino fundamental estão adotando o ensino de programação em sua grade de disciplinas, boa parte delas, ensinando nativamente Python.

Por fim, podemos esperar para o futuro que a linguagem Python cresça exponencialmente, uma vez que novas áreas de atuação como data science e machine learning se popularizem ainda mais. Estudos indicam que para os próximos 10 anos cerca de um milhão de novas vagas surgirão demandando profissionais de tecnologia da área da programação, pode ter certeza que grande parcela desse público serão programadores com domínio em Python.

Python será o limite?

Esta é uma pergunta interessante de se fazer porque precisamos parar uns instantes e pensar no futuro. Raciocine que temos hoje um crescimento exponencial do uso de machine learning, data science, internet das coisas, logo, para o futuro podemos esperar uma demanda cada vez maior de processamento de dados, o que não necessariamente signifique que será mais complexo desenvolver ferramentas para suprir tal demanda.

A versão 3 do Python é bastante robusta e consegue de forma natural já trabalhar com tais tecnologias. Devido a comunidade enorme que desenvolve para Python, podemos esperar que para o futuro haverão novas versões implementando novos recursos de forma natural. Será muito difícil virmos surgir outra linguagem “do zero” ou que tenha usabilidade parecida com Python.

Se você olhar para trás verá uma série de linguagens que foram descontinuadas com o tempo, mesmo seus desenvolvedores insistindo e injetando tempo em dinheiro em seu desenvolvimento elas não eram modernas o suficiente. Do meu ponto de vista não consigo ver um cenário do futuro ao qual Python não consiga se adaptar.

Entenda que na verdade não é uma questão de uma linguagem concorrer contra outra, na verdade independente de qual linguagem formos usar, precisamos de uma que seja capaz de se adaptar ao seu tempo e as nossas necessidades. Num futuro próximo nosso diferencial como profissionais da área de tecnologia será ter conhecimento sobre C#, Java ou Python. Garanto que você já sabe em qual delas estou apostando minhas fichas...

2 – Ambiente de programação

Na linguagem Python, e, não diferente das outras linguagens de programação, quando partimos do campo das ideias para a prática, para codificação/programação não basta que tenhamos um computador rodando seu sistema operacional nativo. É importante começarmos a raciocinar que, a partir do momento que estamos entrando na programação de um sistema, estamos trabalhando com seu backend, ou seja, com o que está por trás das cortinas, com aquilo que o usuário final não tem acesso. Para isto, existe uma gama enorme de softwares e ferramentas que nos irão auxiliar a criar nossos programas e levá-los ao frontend.

Linguagens de alto, baixo nível e de máquina

Seguindo o raciocínio lógico do tópico anterior, agora entramos nos conceitos de linguagens de alto e baixo nível e posteriormente a linguagem de máquina.

Quando estamos falando em linguagens de alto e baixo nível, estamos falando sobre o quanto distante está a sintaxe do usuário. Para ficar mais claro, uma linguagem de alto nível é aquela mais próximo do usuário, que usa termos e conceitos normalmente vindos do inglês e que o usuário pode pegar qualquer bloco de código e o mesmo será legível e fácil de compreender. Em oposição ao conceito anterior, uma linguagem de baixo nível é aquela mais próxima da máquina, com instruções que fazem mais sentido ao interpretador do que ao usuário.

Quando estamos programando em Python estamos num ambiente de linguagem de alto nível, onde usaremos expressões em inglês e uma sintaxe fácil para por fim dar nossas instruções ao computador. Esta linguagem posteriormente será convertida em linguagem de baixo nível e por fim se tornará sequências de instruções para registradores e portas lógicas. Imagine que o comando que você digita para exibir um determinado texto em tela é convertido para um segundo código que o interpretador irá ler como bytecode, convertendo ele para uma linguagem de mais baixo nível chamada Assembly, que irá pegar tais instruções e converter para binário, para que por fim tais instruções virem sequências de chaveamento para portas lógicas do processador.

Nos primórdios da computação se convertiam algoritmos em sequências de cartões perfurados que iriam programar sequências de chaveamento em máquinas ainda valvuladas, com o surgimento dos transistores entramos na era da eletrônica digital onde foi possível miniaturizar os registradores, trabalhar com milhares deles de forma a realizar centenas de milhares de cálculos por segundo. Desenvolvemos linguagens de programação de alto nível para que justamente facilitássemos a leitura e escrita de nossos códigos, porém a linguagem de máquina ainda é, e por muito tempo será, binário, ou seja, sequências de informações de zeros e uns que se convertem em pulsos ou ausência de pulsos elétricos nos transistores do processador.

Ambientes de desenvolvimento integrado

Entendidos os conceitos de linguagens de alto e baixo nível e de máquina, por fim vamos falar sobre os IDEs, sigla para ambiente de desenvolvimento integrado. Já rodando nosso sistema operacional temos a frontend do sistema, ou seja, a capa ao qual o usuário tem acesso aos recursos do mesmo. Para que possamos ter acesso aos bastidores do sistema e por fim programar em cima dele, temos softwares específicos para isto, os chamados IDE's. Nestes ambientes temos as ferramentas necessárias para tanto trabalhar a nível de código quanto para testar o funcionamento de nossos programas no sistema operacional.

As IDEs vieram para integrar todos softwares necessários para esse processo de programação em um único ambiente, já houveram épocas onde se programava separado de onde se compilava, separado de onde se debugava e por fim separado de onde se rodava o programa, entenda que as ides unificam todas camadas necessárias para que possamos nos concentrar em escrever nossos códigos e rodá-los ao final do processo. Entenda que é possível programar em qualquer editor de texto, como o próprio bloco de notas ou em um terminal, o uso de IDEs se dá pela facilidade de possuir todo um espectro de ferramentas dedicadas em um mesmo lugar.

Principais IDEs

Como mencionado no tópico anterior, as IDEs buscam unificar as ferramentas necessárias para facilitar a vida do programador, claro que é possível programar a partir de qualquer bloco de notas, mas visando ter um ambiente completo onde se possa usar diversas ferramentas, testar e compilar seu código, o uso de uma IDE é altamente recomendado.

Existem diversas IDEs disponíveis no mercado, mas basicamente aqui irei recomendar duas delas.

Pycharm – A IDE Pycharm desenvolvida e mantida pela JetBrains, é uma excelente opção no sentido de que possui versão completamente gratuita, é bastante intuitiva e fácil de aprender a usar seus recursos e por fim ela oferece um ambiente com suporte em tempo real ao uso de console/terminal próprio, sendo possível a qualquer momento executar testes nos seus blocos de código.

Disponível em: <https://www.jetbrains.com/pycharm/>

Anaconda – Já a suíte Anaconda, também gratuita, conta com uma série de ferramentas que se destacam por suas aplicabilidades. Python é uma linguagem muito usada para data science, machine learning, e a suíte anaconda oferece softwares onde é possível trabalhar dentro dessas modalidades com ferramentas dedicadas a ela, além, é claro, do próprio ambiente de programação comum, que neste caso é o VisualStudioCode, e o Jupyter que é um terminal que roda via browser.

Disponível em: <https://www.anaconda.com/download/>

Importante salientar também que, é perfeitamente normal e possível programar direto em terminal, seja em Linux, seja em terminais oferecidos pelas próprias IDEs, a escolha de usar um ou outro é de particularidade do programador. É possível inclusive usar um terminal junto ao próprio editor da IDE.

Como dito anteriormente, existem diversas IDEs e ferramentas que facilitarão sua vida como programador, a verdade é que o interessante mesmo você dedicar um tempinho a testar as duas e ver qual você se adapta melhor. Não existe uma IDE melhor que a outra, o que existem são ferramentas que se adaptam melhor as suas necessidades enquanto programador.

3 – Lógica de programação

Como já mencionei em um tópico anterior, uma linguagem de programação é uma linguagem como qualquer outra em essência, o diferencial é que na programação são meios que criamos para conseguir passar comandos a serem executados em um computador. Quando falamos sobre um determinado assunto, automaticamente em nossa língua falada nativa geramos uma sentença que possui uma certa sintaxe e lógica de argumento para que a outra pessoa entenda o que estamos querendo transmitir. Se não nos expressarmos de forma clara podemos não ser entendidos ou o pior, ser mal-entendidos.

Um computador é um mecanismo exato, ele não tem (ainda) a capacidade de discernimento e abstração que possuímos, logo, precisamos passar suas instruções de forma literal e ordenada, para que a execução de um processo seja correta.

Quando estudamos lógica de programação, estudamos métodos de criar sequências lógicas de instrução, para que possamos usar tais sequências para programar qualquer dispositivo para que realize uma determinada função. Essa sequência lógica recebe o nome de algoritmo.

Algoritmos

Todo estudante de computação no início de seu curso recebe essa aula, algoritmos. Algoritmos em suma nada mais é do que uma sequência de passos onde iremos passar uma determinada instrução a ser realizada. Imagine uma receita de bolo, onde há informação de quais ingredientes serão usados e um passo a passo de que ordem cada ingrediente será adicionado e misturado para que no final do processo o bolo dê certo, seja comestível e saboroso.

Um algoritmo é exatamente a mesma coisa, conforme temos que programar uma determinada operação, temos que passar as instruções passo a passo para que o interpretador consiga as executar e chegar ao fim de um processo.

Quando estudamos algoritmos basicamente temos três tipos básicos de algoritmo. Os que possuem uma entrada e geram uma saída, os que não possuem entrada, mas geram saída e os que possuem entrada, mas não geram saída.

Parece confuso, mas calma, imagine que na sua receita de bolo você precise pegar ingrediente por ingrediente e misturar, para que o bolo fique pronto depois de assado (**neste caso temos entradas que geram uma saída**). Em outro cenário imagine que você tem um sachê com a mistura já pronta de ingredientes, bastando apenas adicionar leite e colocar para assar (**neste caso, não temos as entradas, mas temos a saída**). Por fim imagine que você é a empresa que produz o sachê com os ingredientes já misturados e pré-prontos, você tem os ingredientes e produz a mistura, porém não é você que fará o bolo (**neste caso temos as entradas e nenhuma saída**).

Em nossos programas haverão situações onde iremos criar scripts que serão executados ou não de acordo com as entradas necessárias e as saídas esperadas para resolver algum tipo de problema computacional.

Sintaxe em Python

Você provavelmente não se lembra do seu processo de alfabetização, quando aprendeu do zero a escrever suas primeiras letras, sílabas, palavras até o momento em que interpretou sentenças inteiras de acordo com a lógica formal de escrita, e muito provavelmente hoje o faz de forma automática apenas transliterando seus pensamentos. Costumo dizer que aprender uma linguagem de programação é muito parecido, você literalmente estará aprendendo do zero um meio de “se comunicar” com o computador de forma a lhe passar instruções a serem executadas dentro de uma ordem e de uma lógica.

Toda linguagem de programação tem sua maneira de transliterar a lógica de um determinado algoritmo em uma linguagem característica que será interpretada “pelo computador”, isto chamamos de sintaxe. Toda linguagem de programação tem sua sintaxe característica, o programador deve respeitá-la até porque o interpretador é uma camada de software o qual é “programado” para entender apenas o que está sob a sintaxe correta. Outro ponto importante é que uma vez que você domina a sintaxe da linguagem ao qual está trabalhando, ainda assim haverão erros em seus códigos devido a erros de lógica.

Não se preocupe porque a coisa mais comum, até mesmo para programadores experientes é a questão de vez em quando cometer algum pequeno erro de sintaxe, ou não conseguir transliterar seu algoritmo numa lógica correta.

A sintaxe na verdade será entendida ao longo deste livro, até porque cada tópico que será abordado, será algum tipo de instrução que estaremos aprendendo e haverá uma forma correta de codificar tal instrução para que finalmente seja interpretada.

Já erros de lógica são bastante comuns nesse meio e, felizmente ou infelizmente, uma coisa depende da outra. Raciocine que você tem uma ideia, por exemplo, de criar uma simples calculadora. De nada adianta você saber programar e não entender o funcionamento de uma calculadora, ou o contrário, você saber toda lógica de funcionamento de uma calculadora e não ter ideia de como transformar isto em código.

Portanto, iremos bater muito nessa tecla ao longo do livro, tentando sempre entender qual a lógica por trás do código e a codificação de seu algoritmo.

Para facilitar, nada melhor do que usarmos um exemplo prático de erro de sintaxe e de lógica respectivamente.

Supondo que queiramos exibir em tela a soma de dois números. 5 e 2, respectivamente.

*Esses exemplos usam de comandos que serão entendidos detalhadamente em capítulos posteriores, por hora, entenda que existe uma maneira certa de se passar informações para “a máquina”, e essas informações precisam obrigatoriamente ser em uma linguagem que faça sentido para o interpretador da mesma.

Ex 1:

```
print('5' + '2')
```

O resultado será **52**, porque de acordo com a sintaxe do Python 3, tudo o que digitamos entre ‘aspas’ é uma **string** (texto), sendo assim o interpretador pegou o texto ‘5’ e o texto ‘2’ e, como são dois textos, os concatenou de acordo com o símbolo de soma. Em outras palavras, aqui o nosso objetivo de somar 5 com 2 não foi realizado com sucesso porque passamos a informação para “a máquina” de maneira errada. Um erro típico de lógica.

Ex 2:

```
print(5 + 2)
```

O resultado é **7**, já que o interpretador pegou os dois valores inteiros 5 e 2 e os somou de acordo com o símbolo **+**, neste caso um operador matemático de soma. Sendo assim, codificando da maneira correta o interpretador consegue realizar a operação necessária.

Ex 3:

```
Print(5) + 2 =
```

O interpretador irá acusar um erro de sintaxe nesta linha do código pois ele não reconhece **Print** (iniciado em maiúsculo) e também não entende quais são os dados a serem usados e seus operadores. Um erro típico de sintaxe, uma vez que não estamos passando as informações de forma que o interpretador consiga interpretá-las adequadamente.

Em suma, como dito anteriormente, temos que ter bem definida a ideia do que queremos criar, e programar de forma que o interpretador consiga receber e trabalhar com essas informações. Sempre que nos deparamos com a situação de escrever alguma linha ou bloco de código e nosso interpretador acusar algum erro, devemos revisar o código em busca de que tipo de erro foi esse (lógica ou sintaxe) e procurar corrigir no próprio código.

Palavras reservadas

Na linguagem Python existem uma série de palavras reservadas para o sistema, ou seja, são palavras chave que o interpretador busca e usa para receber instruções a partir delas. Para ficar mais claro vamos pegar como exemplo a palavra **print**, em Python **print** é um comando que serve para exibir em tela ou em console um determinado dado ou valor, sendo assim, é impossível criarmos uma variável com nome **print**, pois esta é uma palavra reservada ao sistema.

Ao todo são 31 palavras reservadas na sintaxe.

```
and    del    from    not    while
as     elif   global  or     with
assert else   if      pass   yield
break  except  import  print
class  exec   in     raise
continue finally is     return
def    for    lambda try
```

Repare que todas palavras utilizadas são termos em inglês, como Python é uma linguagem de alto nível, ela usa uma linguagem bastante próxima do usuário, com conhecimento básico de inglês é possível traduzir e interpretar o que cada palavra reservada faz.

À medida que formos progredindo você irá automaticamente associar que determinadas “palavras” são comandos ou instruções internas a linguagem. Você precisa falar uma língua que a máquina possa reconhecer, para que no final das contas suas instruções sejam entendidas, interpretadas e executadas.

Análise léxica

Quando estamos programando estamos colocando em prática o que planejamos anteriormente sob a ideia de um algoritmo. Algoritmos por si só são sequências de instruções que iremos codificar para serem interpretadas e executar uma determinada função. Uma das etapas internas do processo de programação é a chamada análise léxica.

Raciocine que todo e qualquer código deve seguir uma sequência lógica, um passo-a-passo a ser seguido em uma ordem definida, basicamente quando estamos programando dividimos cada passo de nosso código em uma linha nova/diferente. Sendo assim, temos que respeitar a ordem e a sequência lógica dos fatos para que eles sempre ocorram na ordem certa.

O interpretador segue fielmente cada linha e seu conteúdo, uma após a outra, uma vez que “ele” é uma camada de programa sem capacidade de raciocínio e interpretação como nós humanos. Sendo assim devemos criar cada linha ou bloco de código em uma sequência passo-a-passo que faça sentido para o interpretador.

Por fim, é importante ter em mente que o interpretador sempre irá ler e executar, de forma sequencial, linha após linha e o conteúdo de cada linha sempre da esquerda para direita. Por exemplo:

```
print('Seja bem vindo!')  
print('Você é' + 3 * 'muito' + 'legal!')
```

Como mencionamos anteriormente, o interpretador sempre fará a leitura das linhas de cima para baixo. Ex: linha 1, linha 2, linha 3, etc... e o interpretador sempre irá ler o conteúdo da linha da esquerda para direita.

Nesse caso o retorno que o usuário terá é:

Seja bem vindo!

Você é muito muito muito legal.

Repare que pela sintaxe a linha 2 do código tem algumas particularidades, existem 3 **strings** e um operador mandando repetir 3 vezes uma delas, no caso 3 vezes ‘**muito**’, e como são **strings**, o símbolo de + está servindo apenas para concatená-las. Embora isso ainda não faça sentido para você, fique tranquilo pois iremos entender cada ponto de cada linha de código posteriormente.

Indentação

Python é uma linguagem de forte indentação, ou seja, para fácil sintaxe e leitura, suas linhas de código não precisam necessariamente de uma pontuação, mas de uma tabulação correta. Quando linhas/blocos de código são filhos de uma determinada função ou parâmetro, devemos organizá-los de forma a que sua tabulação siga um determinado padrão.

Diferente de outras linguagens de programação que o interpretador percorre cada sentença e busca uma pontuação para demarcar seu fim, em Python o interpretador usa uma indentação para conseguir ver a hierarquia das sentenças.

O interpretador de Python irá considerar linhas e blocos de código que estão situados na mesma tabulação (margem) como blocos filhos do mesmo objeto/parâmetro. Para ficar mais claro, vejamos dois exemplos, o primeiro com indentação errada e o segundo com a indentação correta:

Indentação errada:

```
variavel1 = input('Digite um número: ')
if variavel1 >= str(0):
    print('Número Positivo.')
    print(f'O número é: {variavel1}')
else:
    print('Número Negativo')
    print(f'O número é: {variavel1}')
```

Repare que neste bloco de código não sabemos claramente que linhas de código são filhas e nem de quem... e mais importante que isto, com indentação errada o interpretador não saberá quais linhas ou blocos de código são filhas de quem, não conseguindo executar e gerar o retorno esperado, gerando erro de sintaxe.

Indentação correta:

```
variavel1 = input('Digite um número: ')

if variavel1 >= str(0):
    print('Número Positivo.')
    print(f'O número é: {variavel1}')
else:
    print('Número Negativo')
    print(f'O número é: {variavel1}')
```

Agora repare no mesmo bloco de código, mas com as indentações corretas, podemos ver de acordo com as margens e espaçamentos quais linhas de código são filhas de

quais comandos ou parâmetros.

O mesmo ocorre por parte do interpretador, de acordo com a tabulação usada, ele consegue perceber que inicialmente existe uma variável **variavel1** declarada que pede que o usuário digite um número, também que existem duas estruturas condicionais **if** e **else**, e que dentro delas existem instruções de imprimir em tela o resultado de acordo com o que o usuário digitou anteriormente e foi atribuído a **variavel1**.

Se você está vindo de outra linguagem de programação como **C** ou uma de suas derivadas, você deve estar acostumado a sempre encerrar uma sentença com ponto e vírgula ; Em Python não é necessário uma pontuação, mas se por ventura você inserir, ele simplesmente irá reconhecer que ali naquele ponto e vírgula se encerra uma instrução. Isto é ótimo porque você verá que você volta e meia cometará algumas exceções usando vícios de programação de outras linguagens que em Python ao invés de gerarmos um erro, o interpretador saberá como lidar com essa exceção. Ex:

Código correto:

```
print('Olá Amigo')  
print('Tudo bem?')
```

Código com vício de linguagem:

```
print('Olá Amigo');  
print('Tudo bem?')
```

Código extrapolado:

```
print('Olá Amigo'); print('Tudo bem?')
```

Nos três casos o interpretador irá contornar o que for vício de linguagem e entenderá a sintaxe prevista, executando normalmente os comandos **print()**.

O retorno será: **Olá Amigo**

Tudo bem?

4 – Estrutura básica de um programa

Enquanto em outras linguagens de programação você tem de criar toda uma estrutura básica para que realmente você possa começar a programar, Python já nos oferece praticamente tudo o que precisamos pré-carregado de forma que ao abrirmos uma IDE nossa única preocupação inicial é realmente programar.

Python é uma linguagem “batteries included”, termo em inglês para (pilhas inclusas), ou seja, ele já vem com o necessário para seu funcionamento pronto para uso. Posteriormente iremos implementar novas bibliotecas de funcionalidades em nosso código, mas é realmente muito bom você ter um ambiente de programação realmente pronto para uso.

Que tal começarmos pelo programa mais básico do mundo. Escreva um programa que mostre em tela a mensagem “Olá Mundo”. Fácil não?

Vamos ao exemplo:

```
print('Olá Mundo!!!')
```

Sim, por mais simples que pareça, isto é tudo o que você precisará escrever para que de fato, seja exibida a mensagem **Olá Mundo!!!** na tela do usuário.

Note que existe no início da sentença um **print()**, que é uma palavra reservada ao sistema que tem a função de mostrar algo no console ou na tela do usuário, **print()** sempre será seguido de () parênteses, pois dentro deles estará o que chamamos de argumentos/parâmetros dessa função, neste caso, uma **string** (frase) ‘**Olá Mundo!!!**’, toda frase, para ser reconhecida como tal, deve estar entre aspas. Então fazendo o raciocínio lógico desta linha de código, chamamos a função **print()** que recebe como argumento ‘**Olá Mundo!!!**’.

O retorno será: **Olá Mundo!!!**

Em outras linguagens de programação você teria de importar bibliotecas para que fosse reconhecido mouse, teclado, suporte a entradas e saída em tela, definir um escopo, criar um método que iria chamar uma determinada função, etc... etc... etc... Em Python basta já de início dar o comando que você quer para que ele já possa ser executado. O interpretador já possui pré-carregado todos recursos necessários para identificar uma função e seus métodos, assim como os tipos de dados básicos que usaremos e todos seus operadores.

5 – Tipos de dados

Independentemente da linguagem de programação que você está aprendendo, na computação em geral trabalhamos com dados, e os classificamos conforme nossa necessidade. Para ficar mais claro, raciocine que na programação precisamos separar os dados quanto ao seu tipo. Por exemplo uma **string**, que é o termo reservado para qualquer tipo de dado alfanumérico (qualquer tipo de palavra/texto que contenha letras e números).

Já quando vamos fazer qualquer operação aritmética precisamos tratar os números conforme seu tipo, por exemplo o número **8**, que para programação é um **int** (número inteiro), enquanto o número **8.2** é um **float** (número com casa decimal). O ponto que você precisa entender de imediato é que não podemos misturar tipos de dados diferentes em nossas operações, porque o interpretador não irá conseguir distinguir que tipo de operação você quer realizar uma vez que ele faz uma leitura léxica e “literal” dos dados.

Por exemplo: Podemos dizer que Maria tem 8 anos, e nesse contexto, para o interpretador, o **8** é uma **string**, é como qualquer outra palavra dessa mesma sentença. Já quando pegamos dois números para somá-los por exemplo, o interpretador espera que esses números sejam **int** ou **float**, mas nunca uma **string**.

Parece muito confuso de imediato, mas com os exemplos que iremos posteriormente abordar você irá de forma automática diferenciar os dados quanto ao seu tipo e seu uso correto.

Segue um pequeno exemplo dos tipos de dados mais comuns que usaremos em Python:

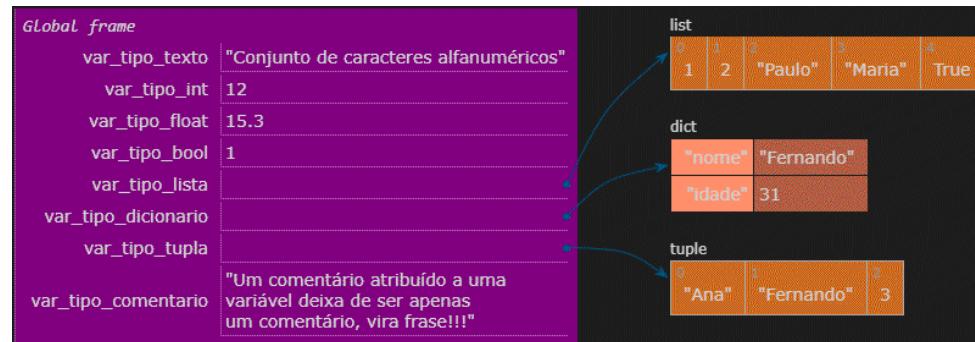
Tipo	Descrição	Exemplo
Int	Número real inteiro, sem casas decimais	12
Float	Número com casas decimais	12.8
Bool	Booleano / Binário (0 ou 1)	0 (ou 1)
String	Texto com qualquer caractere alfanumérico	‘palavra’ ‘marca d’água’
List	Listas(s)	[2, ‘Pedro’, 15.9]
Dict	Dicionário(s)	{‘nome’: ‘João da Silva’, ‘idade’: 32}

Repare também que cada tipo de dado possui uma sintaxe própria para que o interpretador os reconheça como tal. Vamos criar algumas variáveis (que veremos no

capítulo a seguir) apenas para atribuir diferentes tipos de dados, de forma que possamos finalmente visualizar como cada tipo de dados deve ser representado.

```
variavel_tipo_string = 'Conjunto de caracteres alfanuméricos'  
variavel_tipo_int = 12    #número inteiro  
variavel_tipo_float = 15.3  #número com casas decimais  
variavel_tipo_bool = 1    #Booleano / Binário (0 ou 1)  
variavel_tipo_lista = [1, 2, 'Paulo', 'Maria', True]  
variavel_tipo_dicionario = {'nome':'Fernando', 'idade':31,}  
variavel_tipo_tupla = ('Ana', 'Fernando', 3)  
variavel_tipo_comentario = """Um comentário atribuído a uma  
variável deixa de ser apenas  
um comentário, vira frase!!!"""
```

Representação visual:



6 – Comentários

Desculpe a redundância, mas comentários dentro das linguagens de programação servem realmente para comentar determinados blocos de código, para criar anotações sobre o mesmo. É uma prática bastante comum à medida que implementamos novas funcionalidades em nosso código ir comentando-o também, para facilitar nosso entendimento quando revisarmos o mesmo. Essa prática também é bastante comum quando pegamos códigos de domínio público, normalmente o código virá com comentários do seu autor explicando os porquês de determinadas linhas de código.

A sintaxe para comentar nossos códigos em Python é bastante simples, basicamente existem duas maneiras de comentar o código, quando queremos fazer um comentário que não será maior do que uma linha usaremos o símbolo `#` e em seguida o devido comentário. Já quando precisamos fazer algum comentário mais elaborado, que irá ter mais de uma linha, usaremos “”aspas triplas antes de nosso comentário e depois dele para o terminar””.

A ideia de comentar determinados blocos de código é uma maneira do programador colocar anotações sobre determinadas linhas de código. Tenha em mente que o interpretador não lê o que o usuário determinou como comentário, tudo o que estiver após `#` ou entre “” o interpretador irá simplesmente ignorar.

Vamos ao exemplo:

```
nome = 'Maria'  
#Maria é a nova funcionária  
  
print('Bem vinda Maria!!!')  
#acima está uma mensagem de boas-vindas a ela.
```

Exemplo 2:

```
'''Este programa está sendo escrito  
para que Maria, a nova funcionária,  
comece a se integrar com o sistema.'''

```

Neste exemplo acima existem comentários dos dois tipos, o interpretador irá fazer apenas a leitura da variável `nome` e irá executar o comando `print()`, ignorando todo o resto.

Porém se por ventura você quiser fazer com que um comentário passe a ser lido em seu código o mesmo deverá ser associado a uma variável. Ex:

```
'''Exemplo de comentário  
este, inclusive, não será'''
```

lido pelo interpretador"

Comentário interno, não lido pelo interpretador e não visível pelo usuário. Apenas comentário para alguma linha/bloco de código.

```
comentario1 = """Exemplo de comentário  
e agora sim será lido  
pelo interpretador!!!"""
```

```
print(comentario1)
```

Comentário que será exibido ao usuário, pois está associado a uma variável e o comando **print()** está executando sua exibição.

O retorno será: **Exemplo de comentário**
e agora sim será lido
pelo interpretador!!!

Uma prática bastante comum é, enquanto testamos nossos blocos de código usar do artifício de comentário para isolar algum elemento, uma vez que este quando comentado passa a ser ignorado pelo interpretador.

```
var1 = 2019  
var2 = 2020
```

```
soma = var1 + var2
```

```
print(soma)
```

Repare que o código acima por mais básico que seja possui duas variáveis **var1** e **var2**, uma operação de **soma** entre elas e um comando **print()** que irá exibir em tela o resultado de **soma**. Se quisermos por exemplo, apenas a fim de testes, ignorar a operação **soma** declarada no código e realizar esta operação diretamente dentro do comando **print()**, isto é perfeitamente possível. Bastando “comentar” a função **soma**, ela passa a ser ignorada pelo interpretador. Ex:

```
var1 = 2019  
var2 = 2020  
var3 = 2021
```

```
#soma = var1 + var2
```

```
print(var1 + var2)
```

O retorno em ambos os casos será: **4039**

Note que no segundo bloco de código, ao inserir o marcador **#** a frente da função **soma**, a transformamos em um simples comentário, ignorado pelo interpretador, para

traze-la de volta à ativa basta “descomentar” a mesma.

7 – Variáveis / objetos

Uma variável basicamente é um espaço alocado na memória ao qual iremos armazenar um dado, valor ou informação. Imagine que você tem uma escrivaninha com várias gavetas, uma variável é uma dessas gavetas ao qual podemos guardar dentro dela qualquer coisa (qualquer tipo de dado) e ao mesmo tempo ter acesso fácil a este dado durante a execução de nosso programa.

Python é uma linguagem dinamicamente tipada, ou seja, quando trabalhamos com variáveis/objetos (itens aos quais iremos atribuir dados ou valores), podemos trabalhar livremente com qualquer tipo de dado e se necessário, também alterar o tipo de uma variável a qualquer momento.

Outra característica importante de salientar neste momento é que Python, assim como outras linguagens, ao longo do tempo foi sofrendo uma série de mudanças que trouxeram melhorias em sua forma de uso. Na data de publicação deste livro estamos usando a versão 3.7 da linguagem Python, onde se comparado com as versões anteriores da mesma, houve uma série de simplificações em relação a maneira como declaramos variáveis, definimos funções, iteramos dados. Por fim apenas raciocine que não iremos estar nos focando em sintaxe antiga ou que está por ser descontinuada, não há sentido em nos atermos a isto, todo e qualquer código que será usado neste livro respeitará a sintaxe mais atual.

Declarando uma variável

A declaração básica de uma variável/objeto sempre seguirá uma estrutura lógica onde, toda variável deve ter um nome (desde que não seja uma palavra reservada ao sistema) e algo atribuído a ela (qualquer tipo de dado ou valor).

Partindo diretamente para prática, vamos ver um exemplo de declaração de uma variável:

```
variavel1 = 11
```

Neste caso, inicialmente declaramos uma variável de nome **variavel1** que por sua vez está recebendo o valor **10** como valor (o símbolo de **=** aqui é utilizado para atribuir um valor a variável). No contexto de declaração de variáveis, o símbolo de igualdade não está comparando ou igualando os lados, mas está sendo usado para atribuir um dado ou valor a uma variável.

```
variavel1 = 11  
print(variavel1)
```

Aqui, inicialmente apenas para fins de exemplo, na segunda linha do código usamos o comando **print()** que dentro de seus “parênteses” está instanciando a variável que acabamos de criar, a execução dessa linha de código irá exibir em tela para o usuário o dado/valor que for conteúdo, que estiver atribuído a variável **variavel1**.

Neste caso o retorno será: **11**

Conforme você progredir em seus estudos de programação você irá notar que é comum possuirmos várias variáveis, na verdade, quantas forem necessárias em nosso programa. Não existe um limite, você pode usar à vontade quantas variáveis forem necessárias desde que respeite a sua sintaxe e que elas tenham um propósito no código.

Outro ponto importante é que quando atribuímos qualquer dado ou valor a uma variável o tipo de dado é implícito, ou seja, se você por exemplo atribuir a uma variável simplesmente um número **6**, o interpretador automaticamente identifica esse dado como um **int** (dado numérico do tipo inteiro). Se você inserir um ponto **.** seguido de outro número, **5** por exemplo, tornando esse número agora **6.5**, o interpretador automaticamente irá reconhecer esse mesmo dado agora como **float** (número de ponto flutuante).

O mesmo ocorre quando você abre aspas ‘ ‘ para digitar algum dado a ser atribuído a uma variável, automaticamente o interpretador passará a trabalhar com aquele dado o tratando como do tipo **string** (palavra, texto ou qualquer combinação alfanumérica de caracteres). Posteriormente iremos ver que também é possível declarar explicitamente o tipo de um dado e até convertê-lo de um tipo para outro.

Python também é uma linguagem **case sensitive**, ou seja, ela diferencia caracteres maiúsculos de minúsculos, logo, existem certas maneiras de declarar variáveis que são permitidas enquanto outras não, gerando conflito com o interpretador. A grosso modo podemos declarar variáveis usando qualquer letra minúscula e o símbolo “_” underline simples no lugar do espaço.

Exemplo 1: Declarando variáveis corretamente.

```
variavel1 = 'Ana'  
variavel2 = 'Pedro'  
  
variavel_1 = 'Ana'  
variavel_2 = 'Pedro'
```

Ambos os modelos apresentam sintaxe correta, cabe ao usuário escolher qual modo ele considera mais fácil de identificar essa variável pelo nome.

Exemplo 2: Simulando erro, declarando uma variável de forma não permitida.

```
variavel 1 = 'String'  
14 = 'Numero'
```

Nem **variavel 1** nem a **14** serão reconhecidas pelo interpretador como variáveis porque estão escritas de forma não reconhecida pela sintaxe. No primeiro exemplo o espaço entre variável e 1 gera conflito com o interpretador. No segundo exemplo, um número nunca pode ser usado como nome para uma variável.

Embora permitido, não é recomendável usar o nome de uma variável todo em letras maiúsculas, por exemplo:

```
NOME = 'Fernando'
```

*Se você está vindo de outras linguagens de programação para o Python, você deve conhecer o conceito de que quando se declara uma variável com letras maiúsculas ela se torna uma constante, uma variável imutável. Esse conceito não existe para o Python porque nele as variáveis são tratadas como objetos e sempre serão dinâmicas. Por fim, fique tranquilo que usar essa sintaxe é permitida em Python, não muito recomendável, mas você não terá problemas em seu código em função disso.

Outro ponto importante de destacar é que em função do padrão **case sensitive**, ao declarar duas variáveis iguais, uma com caracteres maiúsculos e outra com caracteres minúsculos serão interpretadas como duas variáveis diferentes. Ex:

```
NOME = 'Fernando'  
nome = 'Rafael'
```

Também é permitido, mas não é recomendado o uso de palavras com acentos: Ex:

```
variável = 'Maria'
```

```
cômodos = 3
```

Por fim, raciocine que o interessante é você criar o hábito de criar seus códigos usando as formas mais comuns de se declarar variáveis, deixando apenas para casos especiais essas exceções.

Exemplos de nomenclatura de variáveis permitidos:

```
variavel = 'Ana'
```

```
variavel1 = 'Ana'
```

```
variavel_1 = 'Ana'
```

```
var_num_1 = 'Ana'
```

```
minhavariavel = 'Ana'
```

```
minha_variavel = 'Ana'
```

```
minhaVariavel = 'Ana'
```

Permitidos, mas não recomendados:

```
variável = 'Ana'
```

```
VARIABEL = 'Ana'
```

```
Variavel = 'Ana'
```

Não permitidos, por poder gerar conflitos com o interpretador de sua IDE:

```
1991 = 'Ana'
```

```
minha variavel = 'Ana'
```

```
lvariavel = 'Ana'
```

*Existirão situações onde um objeto será declarado com letra maiúscula inicial, mas neste caso ele não é uma variável qualquer e seu modo de uso será totalmente diferente. Abordaremos esse tema nos capítulos finais do livro.

Vale lembrar também que por convenção é recomendado criar nomes pequenos e que tenham alguma lógica para o programador, se você vai declarar uma variável para armazenar o valor de um número, faz mais sentido declarar uma variável **numero1** do que algo tipo **ag_23421_m_meuNumero...**

Por fim, vale lembrar que, como visto em um capítulo anterior, existem palavras que são reservadas ao sistema, dessa forma você não conseguirá usar como nome de uma variável. Por exemplo **while**, que é uma chamada para uma estrutura condicional, sendo assim, é impossível usar **while** como nome de variável.

Declarando múltiplas variáveis

É possível, para deixar o código mais enxuto, declarar várias variáveis em uma linha, independentemente do tipo de dado a ser recebido, desde que se respeite a sintaxe correta de acordo com o tipo de dado. Por exemplo, o código abaixo:

```
nome = 'Maria'  
idade = 32  
sexo = 'F'  
altura = 1.89
```

Pode ser agrupado em uma linha, da seguinte forma:

```
nome, idade, sexo, altura = 'Maria', 32, 'F', 1.89
```

A ordem será respeitada e serão atribuídos os valores na ordem aos quais foram citados. (primeira variável com primeiro atributo, segunda variável com segundo atributo, etc...)

Declarando múltiplas variáveis (de mesmo tipo)

Quando formos definir várias variáveis, mas que possuem o mesmo valor ou tipo de dado em comum, podemos fazer essa declaração de maneira resumida. Ex:

Método convencional:

```
num1 = 10  
x = 10  
a1 = 10
```

Método resumido:

```
num1 = x = a1 = 10
```

Repare que neste caso, em uma linha de código foram declaradas 3 variáveis, todas associadas com o valor **10**, o interpretador as reconhecerá como variáveis independentes, podendo o usuário fazer o uso de qualquer uma delas isoladamente a qualquer momento. Por exemplo:

```
num1 = x = a1 = 10
```


8 – Funções Básicas

Funções, falando de forma bastante básica, são linhas ou blocos de códigos aos quais executarão uma determinada ação em nosso código, inicialmente trabalharemos com as funções mais básicas que existem, responsáveis por exibir em tela uma determinada resposta e também responsáveis por interagir com o usuário. Funções podem receber parâmetros de execução (ou não), dependendo a necessidade, uma vez que esses parâmetros nada mais são do que as informações de como os dados deverão interagir internamente para realizar uma determinada função.

Pela sintaxe Python, “chamamos” uma função pelo seu nome, logo em seguida, entre () parênteses podemos definir seus parâmetros, instanciar variáveis ou escrever linhas de código à vontade, desde que não nos esqueçamos que este bloco de código fará parte (será de propriedade) desta função...

Posteriormente trataremos das funções propriamente ditas, como criamos funções personalizadas que realizam uma determinada ação, por hora, para conseguirmos dar prosseguimento em nossos estudos, precisamos entender que existem uma série de funções pré-programadas, prontas para uso, com parâmetros internos que podem ser implícitos ou explícitos ao usuário.

As mais comuns delas, **print()** e **input()** respectivamente nos permitirão exibir em tela o resultado de uma determinada ação de um bloco de código, e interagir com o usuário de forma que ele consiga por meio do teclado inserir dados em nosso programa.

Função print()

Quando estamos criando nossos programas, é comum que de acordo com as instruções que programamos, recebamos alguma saída, seja ela alguma mensagem ou até mesmo a realização de uma nova tarefa. Uma das saídas mais comuns é exibirmos, seja na tela para o usuário ou em console (em programas que não possuem uma interface gráfica), uma mensagem, para isto, na linguagem python usamos a função `print()`.

Na verdade anteriormente já usamos ela enquanto estávamos exibindo em tela o conteúdo de uma variável, mas naquele capítulo esse conceito de fazer o uso de uma função estava lá apenas como exemplo e para obtermos algum retorno de nossos primeiros códigos, agora iremos de fato entender o mecanismo de funcionamento deste tipo de função.

Por exemplo:

```
print('Seja bem vindo!!!')
```

Repare na sintaxe: a função `print()` tem como parâmetro (o que está dentro de parênteses) uma **string** com a mensagem **Seja bem vindo!!!**. Todo parâmetro é delimitado por () parênteses e toda **string** é demarcada por ‘ ‘ aspas para que o interpretador reconheça esse tipo de dado como tal.

O retorno dessa linha de código será: **Seja bem vindo!!!**

Função input()

Em todo e qualquer programa é natural que haja interação do usuário com o mesmo, de modo que com algum dispositivo de entrada o usuário dê instruções ou adicione dados. Começando pelo básico, em nossos programas a maneira mais rudimentar de captar os dados do usuário será por intermédio da função **input()**, por meio dela podemos pedir, por exemplo que o usuário digite um dado ou valor, que internamente será atribuído a uma variável. Ex:

```
nome = input('Digite o seu nome: ')
print('Bem Vindo', nome)
```

Inicialmente declaramos uma variável de nome **nome** que recebe como atributo a função **input()** que por sua vez dentro tem uma mensagem para o usuário. Assim que o usuário digitar alguma coisa e pressionar a tecla ENTER, esse dado será atribuído a variável **nome**. Em seguida, a função **print()** exibe em tela uma mensagem definida concatenada ao nome digitado pelo usuário e atribuído a variável **nome**.

O retorno será: **Bem Vindo Fernando**

*Supondo, é claro, que o usuário digitou Fernando.

Apenas um adendo, como parâmetro de nossa função **print()** podemos instanciar múltiplos tipos de dados, inclusive um tipo de dado mais de uma vez, apenas como exemplo, aprimorando o código anterior, podemos adicionar mais de uma **string** ao mesmo exemplo. Ex:

```
nome = input('Digite o seu nome: ')
print('Bem Vindo', nome, '!!!')
```

O retorno será: **Bem Vindo Fernando !!!**

Explorando a função print()

Como mencionado anteriormente, existem muitas formas permitidas de se “escrever” a mesma coisa, e também existe a questão de que a linguagem Python ao longo dos anos foi sofrendo alterações e atualizações, mudando aos poucos sua sintaxe. O importante de se lembrar é que, entre versões se aprimoraram certos pontos da sintaxe visando facilitar a vida do programador, porém, para programadores que de longa data já usavam pontos de uma sintaxe antiga, ela não necessariamente deixou de funcionar a medida que a linguagem foi sendo atualizada.

Raciocine que muitos dos códigos que você verá internet a fora estarão no padrão Python 2, e eles funcionam perfeitamente no Python, então, se você está aprendendo realmente do zero por meio deste livro, fique tranquilo que você está aprendendo com base na versão mais atualizada da linguagem. Se você programava em Python 2 e agora está buscando se atualizar, fique tranquilo também porque a sintaxe que você usava não foi descontinuada, ela gradualmente deixará de ser usada pela comunidade até o ponto de poder ser removida do núcleo do Python, por hora, ambas sintaxes funcionam perfeitamente. Vamos ver isso na prática:

print() básico – Apenas exibindo o conteúdo de uma variável:

```
nome1 = 'Maria'  
print(nome1)
```

Inicialmente declaramos uma variável de nome **nome1** que recebe como atributo '**Maria**', uma **string**. Em seguida exibimos em tela o conteúdo atribuído a **nome1**.

O retorno será: **Maria**

print() básico – Pedindo ao usuário que dê entrada de algum dado:

```
nome1 = input('Digite o seu nome: ')  
print(nome1)
```

Declaramos uma variável de nome **nome1** que recebe como atributo a função **input()** que por sua vez pede ao usuário que digite alguma coisa. Quando o usuário digitar o que for solicitado e pressionar a tecla ENTER, este dado/valor será atribuído a **nome1**. Da mesma forma que antes, por meio da função **print()** exibimos em tela o conteúdo de **num1**.

O retorno será: **Maria**

*supondo que o usuário digitou Maria.

print() intermediário – Usando máscaras de substituição (Sintaxe antiga):

```
nome1 = input('Digite o seu nome: ')
print('Seja bem vindo(a) %s' %(nome1))
```

Da mesma forma que fizemos no exemplo anterior, declaramos uma variável **nome1** e por meio da função **input()** pedimos uma informação ao usuário. Agora, como parâmetro de nossa função **print()** temos uma mensagem (**string**) que reserva dentro de si, por meio do marcador **%s**, um espaço a ser substituído pelo valor existente em **nome1**. Supondo que o usuário digitou Fernando.

O retorno será: **Seja bem vindo(a) Fernando**

Porém existem formas mais sofisticadas de realizar esse processo de interação, e isso se dá por o que chamamos de máscaras de substituição. Máscaras de substituição foram inseridas na sintaxe Python com o intuito de quebrar a limitação que existia de poder instanciar apenas um dado em nossa **string** parâmetro de nossa função **print()**. Com o uso de máscaras de substituição e da função **.format()** podemos inserir um ou mais de um dado/valor a ser substituído dentro de nossos parâmetros de nossa função **print()**. Ex:

```
nome1 = input('Digite o seu nome: ')
print('Seja bem vindo(a) {} !!!'.format(nome1))
```

A máscara **{ }** reserva um espaço dentro da string a ser substituída pelo dado/valor atribuído a **nome1**.

O retorno será: **Seja bem vindo(a) Fernando !!!**

Fazendo o uso de máscaras de substituição, como dito anteriormente, podemos instanciar mais de um dado/valor/variável dentro dos parâmetros de nossa função **print()**. Ex:

```
nome1 = input('Digite o seu nome: ')
msg1 = 'Por favor entre'
msg2 = 'Você é o primeiro a chegar.'
print('Seja bem vindo {}, {}, {}'.format(nome1, msg1, msg2))
```

Repare que dessa vez temos mais duas variáveis **msg1** e **msg2** respectivamente que possuem **strings** como atributos. Em nossa função **print()** criamos uma **string** de mensagem de boas-vindas e criamos 3 máscaras de substituição, de acordo com a ordem definida em **.format()** substituiremos a primeira pelo nome digitado pelo usuário, a segunda e a terceira máscara pelas frases atribuídas a **msg1** e **msg2**. Dessa forma:

O retorno será: **Seja bem vindo Fernando, Por favor entre, Você é o primeiro a chegar.**

print() intermediário – Usando “f strings” (Sintaxe nova/atual):

Apenas dando um passo adiante, uma maneira mais moderna de se trabalhar com máscaras de substituição se dá por meio de **f strings**, a partir da versão 3 do Python se permite usar um simples parâmetro “**f**” antes de uma string para que assim o interpretador subentenda que ali haverão máscaras de substituição a serem trabalhadas, independentemente do tipo de dado. Dessa forma de facilitou ainda mais o uso de máscaras dentro de nossa função **print()** e outras em geral.

Basicamente declaramos um parâmetro “**f**” antes de qualquer outro e podemos instanciar o que quisermos diretamente dentro das máscaras de substituição, desde o conteúdo de uma variável até mesmo operações e funções dentro de funções, mas começando pelo básico, veja o exemplo:

```
nome = input('Digite o seu nome: ')
ap = input('Digite o número do seu apartamento: ')
print(f'Seja bem vinda {nome}, moradora do ap nº {ap}')
```

Supondo que o usuário digitou respectivamente Maria e 33...

O retorno será: **Seja bem vinda Maria, moradora do ap nº 33**

Quando for necessário exibir uma mensagem muito grande, de mais de uma linha, uma forma de simplificar nosso código reduzindo o número de prints a serem executados é usar a quebra de linha dentro de um **print()**, adicionando um **\n** frente ao texto que deverá estar posicionado em uma nova linha. Ex:

```
nome = 'João'
dia_vencimento = 10
valor_fatura = 149.90

print(f'Olá, caro {nome},\n A sua última fatura com vencimento em {dia_vencimento}\n de janeiro,\n no valor de R${valor_fatura} está próxima do vencimento.\n Favor pagar\n até o prazo para evitar multas.')
```

Repare que existe um comando **print()** com uma sentença enorme, que irá gerar 4 linhas de texto de retorno, combinando o uso de máscaras para sua composição.

O retorno será:

Olá, caro João,
A sua última fatura com vencimento em 10 de janeiro,
no valor de R\$149.9 está próxima do vencimento.
Favor pagar até o prazo para evitar multas.

O que você deve ter em mente, e começar a praticar, é que em nossos programas, mesmo os mais básicos, sempre haverá meios de interagir com o usuário, seja exibindo certo conteúdo para o mesmo, seja interagindo diretamente com ele de forma que ele forneça ou manipule os dados do programa.

Lembre-se que um algoritmo pode ter entradas, para execução de uma ou mais funções e gerar uma ou mais saídas. Estas entradas pode ser um dado importado, um link ou um arquivo instanciado, ou qualquer coisa que esteja sendo inserida por um dispositivo de entrada do computador, como o teclado que o usuário digita ou o mouse interagindo com alguma interface gráfica.

Interação entre variáveis

Agora entendidos os conceitos básicos de como fazer o uso de variáveis, como exibir seu conteúdo em tela por meio da função `print()` e até mesmo interagir com o usuário via função `input()`, hora de voltarmos a trabalhar os conceitos de variáveis, aprofundando um pouco mais sobre o que pode ser possível fazer a partir delas.

A partir do momento que declaramos variáveis e atribuímos valores a elas, podemos fazer a interação entre elas (interação entre seus atributos), por exemplo:

```
num1 = 10
num2 = 5.2
soma = num1 + num2
print(soma)
print(f'O resultado é {soma}')
```

Inicialmente criamos uma variável **num1** que recebe como atributo **10** e uma segunda variável **num2** que recebe como atributo **5.2**. Na sequência criamos uma variável **soma** que faz a soma entre **num1** e **num2** realizando a operação instanciando as próprias variáveis. O resultado da soma será guardado em **soma**. A partir daí podemos simplesmente exibir em tela via função `print()` o valor de **soma**, assim como podemos criar uma mensagem mais elaborada usando máscara de substituição. Dessa forma...

O retorno será: **15.2**

Seguindo com o que aprendemos no capítulo anterior, podemos melhorar ainda mais esse código realizando este tipo de operação básica diretamente dentro da máscara de substituição. Ex:

```
num1 = 10
num2 = 5.2
print(f'O resultado é {num1 + num2}')
```

O retorno será: **15.2**

Como mencionado anteriormente, o fato de a linguagem Python ser dinamicamente tipada nos permite a qualquer momento alterar o valor e o tipo de uma variável. Por exemplo, a variável **a** que antes tinha o valor **10 (int)** podemos a qualquer momento alterar para '**Maria**' (**string**). Por exemplo:

```
a = 10
print(a)
```

O resultado será: **10**

```
a = 10  
a = 'Maria'  
print(a)
```

O resultado será: **Maria**.

A ordem de leitura por parte do interpretador o último dado/valor atribuído a variável **a** foi '**Maria**'. Como explicado nos capítulos iniciais, a leitura léxica desse código respeita a ordem das linhas de código, ao alterarmos o dado/valor de uma variável, o interpretador irá considerar a última linha de código a qual se fazia referência a essa variável e seu último dado/valor atribuído.

Por fim, quando estamos usando variáveis dentro de alguns tipos de operadores podemos temporariamente convertê-los para um tipo de dado, ou deixar mais explícito para o interpretador que tipo de dado estamos trabalhando para que não haja conflito. Por exemplo:

```
num1 = 5  
num2 = 8.2  
soma = int(num1) + int(num2)  
print(soma)
```

O resultado será: **13** (sem casas decimais, porque definimos na expressão de **soma** que **num1** e **num2** serão tratados como **int**, número inteiro, sem casas decimais).

Existe a possibilidade também de já deixar especificado de que tipo de dado estamos falando quando o declaramos em uma variável. Por exemplo:

```
num1 = int(5)  
num2 = float(8.2)  
soma = num1 + num2  
print(soma)
```

A regra geral diz que qualquer operação entre um **int** e um **float** resultará em **float**.

O retorno será **13.2**

Como você deve estar reparando, a sintaxe em Python é flexível, no sentido de que haverão várias maneiras de codificar a mesma coisa, deixando a escolha por parte do usuário. Apenas aproveitando o exemplo acima, duas maneiras de realizar a mesma operação de soma dos valores atribuídos as respectivas variáveis.

```
num1 = 5  
num2 = 8.2  
soma = int(num1) + int(num2)
```

```
# Mesmo que:  
num1 = int(5)  
num2 = float(8.2)  
soma = num1 + num2
```

Por fim, é possível “transformar” de um tipo numérico para outro apenas alterando a sua declaração. Por exemplo:

```
num1 = int(5)  
num2 = float(5)  
print(num1)  
print(num2)
```

O retorno será: **5**

5.0

Note que no segundo retorno, o valor **5** foi declarado e atribuído a **num2** como do tipo **float**, sendo assim, ao usar esse valor, mesmo inicialmente ele não ter sido declarado com sua casa decimal, a mesma aparecerá nas operações e resultados.

```
num1 = int(5)  
num2 = int(8.2)  
soma = num1 + num2  
print(soma)
```

Mesmo exemplo do anterior, mas agora já especificamos que o valor de **num2**, apesar de ser um número com casa decimal, deve ser tratado como inteiro, e sendo assim:

O Retorno será: **13**

E apenas concluindo o raciocínio, podemos aprimorar nosso código realizando as operações de forma mais eficiente, por meio de **f strings**. Ex:

```
num1 = int(5)  
num2 = int(8.2)  
print(f'O resultado da soma é: {num1 + num2}')
```

O Retorno será: **13**

Conversão de tipos de dados

É importante entendermos que alguns tipos de dados podem ser “misturados” enquanto outros não, quando os atribuímos a nossas variáveis e tentamos realizar interações entre as mesmas. Porém existem recursos para elucidar tanto ao usuário quanto ao interpretador que tipo de dado é em questão, assim como podemos, conforme nossa necessidade, convertê-los de um tipo para outro.

A forma mais básica de se verificar o tipo de um dado é por meio da função `type()`. Ex:

```
numero = 5  
print(type(numero))
```

O resultado será: `<class 'int'>`

O que será exibido em tela é o tipo de dado, neste caso, um **int**.

Declarando a mesma variável, mas agora atribuindo **5** entre aspas, pela sintaxe, 5, mesmo sendo um número, será lido e interpretado pelo interpretador como uma **string**. Ex:

```
numero = '5'  
print(type(numero))
```

O resultado será: `<class 'str'>`

Repare que agora, respeitando a sintaxe, ‘**5**’ passa a ser uma **string**, um “texto”.

Da mesma forma, sempre respeitando a sintaxe, podemos verificar o tipo de qualquer dado para nos certificarmos de seu tipo e presumir que funções podemos exercer sobre ele. Ex:

```
numero = [5]  
print(type(numero))
```

O retorno será: `<class 'list'>`

```
numero = {5}  
print(type(numero))
```

O retorno será: `<class 'set'>`

*Lembre-se que a conotação de chaves {} para um tipo de dado normalmente faz dele um dicionário, nesse exemplo acima o tipo de dado retornado foi ‘set’ e não ‘dict’ em

Seguindo esta lógica e, respeitando o tipo de dado, podemos evitar erros de interpretação fazendo com que todo dado ou valor atribuído a uma variável já seja

identificado como tal. Ex:

```
frase1 = str('Raquel tem 15 anos')  
  
print(type(frase1))
```

Neste caso antes mesmo de atribuir um dado ou valor a variável **frase1** já especificamos que todo dado contido nela é do tipo **string**. Executando o comando **print(type(frase))** o retorno será: <class 'str'>

De acordo com o tipo de dados certas operações serão diferentes quanto ao seu contexto, por exemplo tendo duas frases atribuídas às suas respectivas variáveis, podemos usar o operador + para concatená-las (como são textos, soma de textos não existe, mas sim a junção entre eles). Ex:

```
frase1 = str('Raquel tem 15 anos, ')  
frase2 = str('de verdade')  
  
print(frase1 + frase2)
```

O resultado será: **Raquel tem 15 anos, de verdade.**

Já o mesmo não irá ocorrer se misturarmos os tipos de dados, por exemplo:

```
frase1 = str('Raquel tem ')  
frase2 = int(15)  
frase3 = 'de verdade'  
  
print(frase1 + frase2 + frase3)
```

O retorno será um erro de sintaxe, pois estamos tentando juntar diferentes tipos de dados.

Corrigindo o exemplo anterior, usando as 3 variáveis como de mesmo tipo o comando **print** será executado normalmente.

```
frase1 = str('Raquel tem ')  
frase2 = str(15)  
frase3 = ' de verdade'  
  
print(frase1 + frase2 + frase3)
```

O retorno será: **Raquel tem 15 de verdade.**

Apenas por curiosidade, repare que o código apresentado nesse último exemplo não necessariamente está usando f strings porque a maneira mais prática de o executar é instanciando diretamente as variáveis como parâmetro em nossa função **print()**. Já que podemos optar por diferentes opções de sintaxe, podemos perfeitamente fazer o uso da qual considerarmos mais prática.

```
print(frase1 + frase2 + frase3)
# Mesmo que:
print(f'{frase1 + frase2 + frase3}')
```

Nesse exemplo em particular o uso de f strings está aumentando nosso código em alguns caracteres desnecessariamente.

Em suma, sempre preste muita atenção quanto ao tipo de dado e sua respectiva sintaxe, **5** é um **int** enquanto '**5**' é uma **string**. Se necessário, converta-os para o tipo de dado correto para evitar erros de interpretação de seu código.

9 – Operadores

Operadores de Atribuição

Em programação trabalharemos com variáveis/objetos que nada mais são do que espaços alocados na memória onde iremos armazenar dados, para durante a execução de nosso programa, fazer o uso deles. Esses dados, independentemente do tipo, podem receber uma nomenclatura personalizada e particular que nos permitirá ao longo do código os referenciar ou incorporar dependendo a situação. Para atribuir um determinado dado/valor a uma variável teremos um operador que fará esse processo.

A atribuição padrão de um dado para uma variável, pela sintaxe do Python, é feita através do operador `=`, repare que o uso do símbolo de igual (`=`) usado uma vez tem a função de atribuidor, já quando realmente queremos usar o símbolo de igual para igualar operandos, usaremos ele duplicado (`==`). Por exemplo:

```
salario = 955
```

Nesta linha de código temos declaramos a variável **salario** que recebe como valor **955**, esse valor nesse caso é fixo, e sempre que referenciarmos a variável **salario** o interpretador usará seu valor atribuído, **955**.

Uma vez que temos um valor atribuído a uma variável podemos também realizar operações que a referenciem, por exemplo:

```
salario = 955  
aumento1 = 27
```

```
print(salario + aumento1)
```

O resultado será **982**, porque o interpretador pegou os valores das variáveis **salario** e **aumento1** e os somou.

Por fim, também é possível fazer a atualização do valor de uma variável, por exemplo:

```
mensalidade = 229  
mensalidade = 229 + 10  
  
print(mensalidade)
```

O resultado será **239**, pois o último valor atribuído a variável **mensalidade** era **229 + 10**.

Aproveitando o tópico, outra possibilidade que temos, já vimos anteriormente, e na verdade trabalharemos muito com ela, é a de solicitar que o usuário digite algum dado ou algum valor que será atribuído a variável, podendo assim, por meio do operador de atribuição, atualizar o dado ou valor declarado inicialmente, por exemplo:

```
nome = 'sem nome'  
idade = 0  
  
nome = input('Por favor, digite o seu nome: ')  
idade = input('Digite a sua idade: ')  
  
print(nome, idade)
```

Inicialmente as variáveis **nome** e **idade** tinham valores padrão pré-definidos, ao executar esse programa será solicitado que o usuário digite esses dados. Supondo que o usuário digitou **Fernando**, a partir deste momento a variável **nome** passa a ter como valor **Fernando**. Na sequência o usuário quando questionado sobre sua idade irá digitar números, supondo que digitou **33**, a variável **idade** a partir deste momento passa a ter como atribuição **33**. Internamente ocorre a atualização dessa variável para esses novos dados/valores atribuídos.

O retorno será: **Fernando 33**

Atribuições especiais

Atribuição Aditiva:

```
variavel1 = 4  
variavel1 = variavel1 + 5
```

Mesmo que:

```
variavel1 += 5  
  
print(variavel1)
```

Com esse comando o usuário está acrescentando **5** ao valor de **variavel1** que inicialmente era **4**. Sendo $4 + 5$:

O resultado será **9**.

Atribuição Subtrativa:

```
variavel1 = 4  
variavel1 = variavel1 - 3
```

Mesmo que:

```
variavel1 -= 3  
  
print(variavel1)
```

Nesse caso, o usuário está subtraindo 3 de **variavel1**. Sendo $4 - 3$:

O resultado será **1**.

Atribuição Multiplicativa:

```
variavel1 = 4  
variavel1 = variavel1 * 2
```

Mesmo que:

```
variavel1 *= 2  
  
print(variavel1)
```

Nesse caso, o usuário está multiplicando o valor de **variavel1** por 2. Logo 4×2 :

O resultado será: **8**

Atribuição Divisiva:

```
variavel1 = 4
variavel1 = variavel1 / 4

# Mesmo que:
variavel1 /= 4

print(variavel1)
```

Nesse caso, o usuário está dividindo o valor de **variavel1** por 4. Sendo $4 / 4$.
O resultado será: **1**

Módulo de (ou resto da divisão de):

```
variavel1 = 4
variavel1 = variavel1 % 4

# Mesmo que:
variavel1 %= 4

print(variavel1)
```

Será mostrado apenas o resto da divisão de **variavel1** por 4.
O resultado será: **0**

Exponenciação:

```
variavel1 = 4
variavel1 = variavel1 ** 8

# Mesmo que:
variavel1 **= 8

print(variavel1)
```

Nesse caso, o valor de a será multiplicado 8 vezes por ele mesmo. Como a valia 4 inicialmente, a exponenciação será $(4 * 4 * 4 * 4 * 4 * 4 * 4 * 4)$.
O resultado será: **65536**

Divisão Inteira:

```
variavel1 = 512
variavel1 = variavel1 // 512

# Mesmo que:
```

```
variavel1 //= 256
```

```
print(variavel1)
```

Neste caso a divisão retornará um número inteiro (ou arredondado). Ex: 512/256.
O resultado será: **2**

Operadores aritméticos

Os operadores aritméticos, como o nome sugere, são aqueles que usaremos para realizar operações matemáticas em nossos blocos de código. O Python por padrão já vem com bibliotecas pré-alocadas que nos permitem a qualquer momento fazer operações matemáticas simples como soma, subtração, multiplicação e divisão. Para operações de maior complexidade também é possível importar bibliotecas externas que irão implementar tais funções. Por hora, vamos começar do início, entendendo quais são os operadores que usaremos para realizarmos pequenas operações matemáticas.

Operador	Função
+	Realiza a soma de dois números
-	Realiza a subtração de dois números
*	Realiza a multiplicação de dois números
/	Realiza a divisão de dois números

Como mencionei anteriormente, a biblioteca que nos permite realizar tais operações já vem carregada quando iniciamos nosso IDE, nos permitindo a qualquer momento realizar os cálculos básicos que forem necessários. Por exemplo:

Soma:

```
print(5 + 7)
```

O resultado será: 12

Subtração:

```
print(12 - 3)
```

O resultado será: 9

Multiplicação:

```
print(5 * 7)
```

O resultado será: 35

Divisão:

```
print(120 / 6)
```

O resultado será: 20

Operações com mais de 3 operandos:

```
print(5 + 2 * 7)
```

O resultado será **19** porque pela regra matemática primeiro se fazem as multiplicações e divisões para depois efetuar as somas ou subtrações, logo 2×7 são 14 que somados a 5 se tornam 19.

Operações dentro de operações:

```
print((5 + 2) * 7)
```

O resultado será 49 porque inicialmente é realizada a operação dentro dos parênteses $(5 + 2)$ que resulta 7 e aí sim este valor é multiplicado por 7 fora dos parênteses.

Exponenciação:

```
print(3 ** 5) #3 elevado a 5a potência
```

O resultado será **243**, ou seja, $3 \times 3 \times 3 \times 3 \times 3$.

Outra operação possível é a de fazer uma divisão que retorne um número inteiro, “arredondado”, através do operador `//`. Ex:

```
print(9.4 // 3)
```

O resultado será **3.0**, um valor arredondado.

Por fim também é possível obter somente o resto de uma divisão fazendo o uso do operador `%`. Por exemplo:

```
print(10 % 3)
```

O resultado será **1**, porque 10 divididos por 3 são 9 e seu resto é 1.

Apenas como exemplo, para encerrar este tópico, é importante você raciocinar que os exemplos que dei acima poderiam ser executados diretamente no console/terminal de sua IDE, mas claro que podemos usar tais operadores dentro de nossos blocos de código, inclusive atribuindo valores numéricos a variáveis e realizando operações entre elas. Por exemplo:

```
numero1 = 12
```

```
numero2 = 3
```

```
print(numero1 + numero2)
```

```
# Mesmo que:
```

```
print('O resultado da soma é:', numero1 + numero2)
```

Que pode ser aprimorado para:

```
print(f'O resultado da soma é: {numero1 + numero2}')
```

O resultado será **15**, uma vez que **numero1** tem como valor atribuído 12, e **numero2** tem como valor atribuído 3. Somando as duas variáveis chegamos ao valor 15.

Exemplos com os demais operadores:

```
x = 5
```

```
y = 8
```

```
z = 13.2
```

```
print(x + y)
```

```
print(x - y)
```

```
print(x ** z)
```

```
print(z // y)
```

```
print(z / y)
```

Os resultados serão: **13**

-3

1684240309.400895

1.0

1.65

Operadores Lógicos

Operadores lógicos seguem a mesma base lógica dos operadores relacionais, inclusive nos retornando **True** ou **False**, mas com o diferencial de suportar expressões lógicas de maior complexidade.

Por exemplo, se executarmos diretamente no console a expressão **7 != 3** teremos o valor **True** (7 diferente de 3, verdadeiro), mas se executarmos por exemplo **7 != 3 and 2 > 3** teremos como retorno **False** (7 é diferente de 3, mas 2 não é maior que 3, e pela tabela verdade isso já caracteriza **False**).

```
print(7 != 3)
```

O retorno será: **True**

Afinal, 7 é diferente de 3.

```
print(7 != 3 and 2 > 3)
```

O retorno será: **False**

7 é diferente de 3 (Verdadeiro) e (and) 2 é maior que 3 (Falso).

Tabela verdade

Tabela verdade AND (E)

Independentemente de quais expressões forem usadas, se os resultados forem:

$$V \text{ E } V = V$$

$$V \text{ E } F = F$$

$$F \text{ E } V = F$$

$$F \text{ E } F = F$$

No caso do exemplo anterior, **7 era diferente de 3** (V) mas **2 não era maior que 3** (F), o retorno foi **False**.

Neste tipo de tabela verdade, bastando **uma** das proposições ser Falsa para que invalide todas as outras Verdadeiras. Ex:

$$V \text{ e } V = V$$

$$V \text{ e } F \text{ e } V \text{ e } V \text{ e } V \text{ e } V = F$$

Em python o operador "**and**" é um operador lógico (assim como os aritméticos) e pela sequência lógica em que o interpretador trabalha, a expressão é lida da seguinte forma:

```
7 != 3 and 2 > 3
```

```
True and False
```

```
False
```

```
# V e F = F
```

Analisando estas estruturas lógicas estamos tentando relacionar se **7 é diferente de 3** (Verdadeiro) e se **2 é maior que 3** (Falso), logo pela tabela verdade Verdadeiro e Falso resultará **False**.

Mesma lógica para operações mais complexas, e sempre respeitando a tabela verdade.

```
7 != 3 and 3 > 1 and 6 == 6 and 8 >= 9
```

```
True and True and True and False
```

```
False
```

7 diferente de 3 (V) E 3 maior que 1 (V) E 6 igual a 6 (V) E 8 maior ou igual a 9 (F).

É o mesmo que **True and True and True and False**.

Retornando **False** porque uma operação False já invalida todas as outras verdadeiras.

Tabela Verdade OR (OU)

Neste tipo de tabela verdade, mesmo tendo uma proposição Falsa, ela não invalida a Verdadeira. Ex:

$$V \text{ e } V = V$$

$$V \text{ e } F = V$$

$$F \text{ e } V = V$$

$$F \text{ e } F = F$$

Independentemente do número de proposições, bastando ter uma delas verdadeira já valida a expressão inteira.

$$V \text{ e } V = V$$

$$F \text{ e } F = F$$

$$F \text{ e } F \text{ e } V \text{ e } F = V$$

$$F \text{ e } F = F$$

Tabela Verdade XOR (OU Exclusivo/um ou outro)

Os dois do mesmo tipo de proposição são falsos, e nenhum é falso também.

$$V \text{ e } V = F$$

$$V \text{ e } F = V$$

$$F \text{ e } V = V$$

$$F \text{ e } F = F$$

Tabela de Operador de Negação (unário)

not True = F

O mesmo que dizer: Se não é verdadeiro então é falso.

not False = V

O mesmo que dizer: Se não é falso então é verdadeiro.

Também visto como:

not 0 = True

O mesmo que dizer: Se não for zero / Se é diferente de zero então é verdadeiro.

not 1 = False

O mesmo que dizer: Se não for um (ou qualquer valor) então é falso.

Bit-a-bit

O interpretador também pode fazer uma comparação bit-a-bit da seguinte forma:

AND Bit-a-bit

3 = 11 (3 em binário)

2 = 10 (2 em binário)

_ = 10

OR bit-a-bit

3 = 11

2 = 10

_ = 11

XOR bit-a-bit

3 = 11

2 = 10

_ = 01

Por fim, vamos ver um exemplo prático do uso de operadores lógicos, para que faça mais sentido.

```
saldo = 1000
```

```
salario = 4000
```

```
despesas = 2967
```

```
meta = saldo > 0 and salario - despesas >= 0.2 * salario
```

Analisando a variável **meta**: Ela verifica se **saldo** é maior que zero e se **salario** menos **despesas** é maior ou igual a 20% do **salário**.

O retorno será **True** porque o **saldo** era maior que zero e o valor de **salario** menos as **despesas** era maior ou igual a 20% do **salário**. (todas proposições foram verdadeiras).

Operadores de membro

Ainda dentro de operadores podemos fazer consulta dentro de uma lista obtendo a confirmação (**True**) ou a negação (**False**). Por Exemplo:

```
lista = [1, 2, 3, 'Ana', 'Maria']
```

```
print(2 in lista)
```

O retorno será: **True**

Lembrando que uma lista é definida por [] e seus valores podem ser de qualquer tipo, desde que separados por vírgula.

Ao executar o comando **2 in lista**, você está perguntando ao interpretador: "2" é membro desta lista? Se for (em qualquer posição) o retorno será **True**.

Também é possível fazer a negação lógica, por exemplo:

```
lista = [1, 2, 3, 'Ana', 'Maria']
```

```
print('Maria' not in lista)
```

O Retorno será **False**. 'Maria' não está na lista? A resposta foi **False** porque 'Maria' está na lista.

Operadores relacionais

Operadores relacionais basicamente são aqueles que fazem a comparação de dois ou mais operandos, possuem uma sintaxe própria que também deve ser respeitada para que não haja conflito com o interpretador.

- > - Maior que
- \geq - Maior ou igual a
- < - Menor que
- \leq - Menor ou igual a
- $=$ - Igual a
- \neq - Diferente de

O retorno obtido no uso desses operadores será Verdadeiro (**True**) ou Falso (**False**).

Usando como referência o console, operando diretamente nele, ou por meio de nossa função **print()**, sem declarar variáveis, podemos fazer alguns experimentos.

```
print(3 > 4)  
#(3 é maior que 4?)
```

O resultado será **False**. 3 não é maior que 4.

```
print(7 >= 3)  
#(7 é maior ou igual a 3?)
```

O resultado será **True**. 7 é maior ou igual a 3, neste caso, maior que 3.

```
print(3 >= 3)  
#(3 é maior ou igual a 3?)
```

O resultado será **True**. 3 não é maior, mas é igual a 3.

Operadores usando variáveis

```
x = 2
```

```
y = 7
```

```
z = 5
```

```
print(x > z)
```

O retorno será **False**. Porque **x** (2) não é maior que **z** (5).

```
x = 2
```

```
y = 7
```

```
z = 5
```

```
print(z <= y)
```

O retorno será **True**. Porque **z** (5) não é igual, mas menor que **y** (7).

```
x = 2
```

```
y = 7
```

```
z = 5
```

```
print(y != x)
```

O retorno será **True** porque **y** (7) é diferente de **x** (2).

Operadores usando condicionais

```
if (2 > 1):  
    print('2 é maior que 1')
```

Repare que esse bloco de código se iniciou com um **if**, ou seja, com uma condicional, se dois for maior do que um, então seria executado a linha de código abaixo, que exibe uma mensagem para o usuário: **2 é maior que 1**.

Mesmo exemplo usando valores atribuídos a variáveis e aplicando estruturas condicionais (que veremos em detalhe no capítulo seguinte):

```
num1 = 2  
num2 = 1  
  
if num1 > num2:  
    print('2 é maior que 1')
```

O retorno será: **2 é maior que 1**

Operadores de identidade

Seguindo a mesma lógica dos outros operadores, podemos confirmar se diferentes objetos tem o mesmo dado ou valor atribuído. Por exemplo:

```
aluguel = 250  
energia = 250  
agua = 65
```

```
print(aluguel is energia)
```

O retorno será **True** porque os valores atribuídos são os mesmos (nesse caso, **250**).

10 – Estruturas condicionais

Quando aprendemos sobre lógica de programação e algoritmos, era fundamental entendermos que toda ação tem uma reação (mesmo que apenas interna ao sistema), dessa forma, conforme transliterávamos ideias para código, a coisa mais comum era nos depararmos com tomadas de decisão, que iriam influenciar os rumos da execução de nosso programa.

Muitos tipos de programas se baseiam em metodologias de estruturas condicionais, são programadas todas possíveis tomadas de decisão que o usuário pode ter e o programa executa e retorna certos aspectos conforme o usuário vai aderindo a certas opções.

Lembre-se das suas aulas de algoritmos, digamos que, apenas por exemplo o algoritmo **ir_ate_o_mercado** está sendo executado, e em determinada seção do mesmo existam as opções: **SE** estiver chovendo vá pela rua nº1, **SE NÃO** estiver chovendo, continue na rua nº2. Esta é uma tomada de decisão onde o usuário irá aderir a um rumo ou outro, mudando as vezes totalmente a execução do programa, desde que essas possibilidades estejam programadas. Não existe como o usuário tomar uma decisão que não está condicionada ao código, logo, todas possíveis tomadas de decisão dever ser programadas de forma lógica e responsiva.

Ifs, elifs e elses

Uma das partes mais legais de programação, sem sombra de dúvidas, é quando começamos a lidar com estruturas condicionais. Uma coisa é você ter um programa linear, que apenas executa uma tarefa após a outra, sem grandes interações e desfecho sempre linear, como um script passo-a-passo. Já outra coisa é você colocar condições, onde de acordo com as variáveis o programa pode tomar um rumo ou outro.

Como sempre, começando pelo básico, em Python a sintaxe para trabalhar com condicionais é bastante simples se comparado a outras linguagens de programação, basicamente temos os comandos **if** (se), **elif** (o mesmo que **else if** / mas se) e **else** (se não) e os usaremos de acordo com nosso algoritmo demandar tomadas de decisão.

A lógica de execução sempre se dará dessa forma, o interpretador estará executando o código linha por linha até que ele encontrará uma das palavras reservadas mencionadas anteriormente que sinaliza que naquele ponto existe uma tomada de decisão, de acordo com a decisão que o usuário indicar, ou de acordo com a validação de algum parâmetro, o código executará uma instrução, ou não executará nada, ignorando esta condição e pulando para o bloco de código seguinte.

Partindo pra prática:

```
a = 33  
b = 34  
c = 35  
  
if b > a:  
    print('b é MAIOR que a')
```

Declaradas três variáveis **a**, **b** e **c** com seus respectivos valores já atribuídos na linha abaixo existe a expressão **if**, uma tomada de decisão, sempre um **if** será seguido de uma instrução, que se for verdadeira, irá executar um bloco de instrução indentado a ela. Neste caso, se **b** for maior que **a** será executado o comando **print()**.

O retorno será: **b é MAIOR que a**

*Caso o valor atribuído a **b** fosse menor que **a**, o interpretador simplesmente iria pular para o próximo bloco de código.

```
a = 33  
b = 33  
c = 35  
  
if b > a:  
    print('b é MAIOR que a')
```

```
elif b == a:  
    print('b é IGUAL a a')
```

Repare que agora além da condicional **if** existe uma nova condicional declarada, o **elif**. Seguindo o método do interpretador, primeiro ele irá verificar se a condição de **if** é verdadeira, como não é, ele irá pular para esta segunda condicional. Por convenção da segunda condicional em diante se usa **elif**, porém se você usar **if** repetidas vezes, não há problema algum. Seguindo com o código, a segunda condicional coloca como instrução que se **b** for igual a **a**, e repara que nesse caso é, será executado o comando **print**.

O retorno será: **b é IGUAL a a**

```
a = 33  
b = 1  
c = 608  
  
if b > a:  
    print('b é MAIOR que a')  
elif b == a:  
    print('b é IGUAL a a')  
else:  
    print('b é MENOR que a')
```

Por fim, o comando **else** funciona como última condicional, ou uma condicional que é acionada quando nenhuma das condições anteriores do código for verdadeira, **else** pode ter um bloco de código próprio porém note que ele não precisa de nenhuma instrução, já que seu propósito é justamente apenas mostrar que nenhuma condicional (e sua instrução) anterior foi válida. Ex:

```
var1 = 18  
var2 = 2  
var3 = 'Maria'  
var4 = 4  
  
if var2 > var1:  
    print('A segunda variável é maior que a primeira')  
elif var2 == 500:  
    print('A segunda variável vale 500')  
elif var3 == var2:  
    print('A variável 3 tem o mesmo valor da variável 2')  
elif var4 is str('4'):  
    print('A variável 4 não é do tipo string')  
else:  
    print('Nenhuma condição é verdadeira')
```

*Como comentamos rapidamente lá no início do livro, sobre algoritmos, estes podem ter uma saída ou não, aqui a saída é mostrar ao usuário esta mensagem de erro, porém se este fosse um código interno não haveria a necessidade dessa mensagem como retorno, supondo que essas condições simplesmente não fossem válidas o interpretador iria pular para o próximo bloco de código executando o que viesse a seguir.

Por fim, revisando os códigos anteriores, declaramos várias variáveis, e partir delas colocamos suas respectivas condições, onde de acordo com a validação destas condições, será impresso na tela uma mensagem. Importante entendermos também que o interpretador lê a instrução de uma condicional e se esta for verdadeira, ele irá executar o bloco de código e encerrar seu processo ali, pulando a verificação das outras condicionais.

Em suma, o interpretador irá verificar a primeira condicional, se esta for falsa, irá verificar a segunda e assim por diante até encontrar uma verdadeira, ali será executado o bloco de código indentado a ela e se encerrará o processo.

O legal é que como python é uma linguagem de programação dinamicamente tipada, você pode brincar à vontade alterando o valor das variáveis para verificar que tipo de mensagem aparece no seu terminal. Também é possível criar cadeias de tomada de decisão com inúmeras alternativas, mas sempre lembrando que pela sequência lógica em que o interpretador faz sua leitura é linha-a-linha, quando uma condição for verdadeira o algoritmo encerra a tomada de decisão naquele ponto.

And e Or dentro de condicionais

Também é possível combinar o uso de operadores **and** e **or** para elaborar condicionais mais complexas (de duas ou mais condições válidas). Por exemplo:

```
a = 33  
b = 34  
c = 35  
  
if a > b and c > a:  
    print('a é maior que b e c é maior que a')
```

Se **a** for maior que **b** e **c** for maior que **a**:

O retorno será: **a é maior que b e c é maior que a**

```
a = 33  
b = 34  
c = 35  
  
if a > b or a > c:  
    print('a é a variavel maior')
```

Se **a** for maior que **b** ou **a** for maior que **c**.

O retorno será: **a é a variavel maior**

Outro exemplo:

```
nota = int(input('Informe a nota: '))  
  
if nota >= 9:  
    print('Parabéns, quadro de honra')  
elif nota >= 7:  
    print('Aprovado')  
elif nota >= 5:  
    print('Recuperação')  
else:  
    print('Reprovado')
```

Primeiro foi declarada uma variável de nome **nota**, onde será solicitado ao usuário que a atribua um valor, após o usuário digitar uma nota e apertar ENTER, o interpretador fará a leitura do mesmo e de acordo com as condições irá imprimir a mensagem adequada a situação.

Se **nota** for maior ou igual a **9**:

O retorno será: **Parabéns, quadro de honra**

Se **nota** for maior ou igual a **7**:

O retorno será: **Aprovado**

Se **nota** for maior ou igual a **5**:

O retorno será: **Recuperação**

Se **nota** não corresponder a nenhuma das proposições anteriores:

O retorno será: **Reprovado**

Condicionais dentro de condicionais

Outra prática comum é criar cadeias de estruturas condicionais, ou seja, blocos de código com condicionais dentro de condicionais. Python permite este uso e tudo funcionará perfeitamente desde que usada a sintaxe e indentação correta. Ex:

```
var1 = 0
var2 = int(input('Digite um número: '))

if var2 > var1:
    print('Número maior que ZERO')
    if var2 == 1:
        print('O número digitado foi 1')
    elif var2 == 2:
        print('O número digitado foi 2')
    elif var2 == 3:
        print('O número digitado foi 3')
    else:
        print('O número digitado é maior que 3')

else:
    print('Número inválido')
```

Repare que foram criadas 2 variáveis **var1** e **var2**, a primeira já com o valor atribuído **0** e a segunda será um valor que o usuário digitar conforme solicitado pela mensagem, convertido para **int**. Em seguida foi colocada uma estrutura condicional onde se o valor de **var2** for maior do que **var1**, será executado o comando **print** e em seguida, dentro dessa condicional que já foi validada, existe uma segunda estrutura condicional, que com seus respectivos **ifs**, **elifs** e **elses** irá verificar que número é o valor de **var2** e assim irá apresentar a respectiva mensagem. Supondo que o usuário digitou 3:

O retorno será: **Número maior que ZERO**

O número digitado foi 2

Fora dessa cadeia de condicionais (repare na indentação), ainda existe uma condicional **else** para caso o usuário digite um número inválido (um número negativo ou um caractere que não é um número).

Simulando switch/case

Quem está familiarizado com outras linguagens de programação está acostumado a usar a função **Switch** para que, de acordo com a necessidade, sejam tomadas certas decisões. Em Python nativamente não temos a função **switch**, porém temos como simular sua funcionalidade através de uma função onde definiremos uma variável com dicionário dentro, e como um dicionário trabalha com a lógica de **chave:valor**, podemos simular de acordo com uma **opção:umaopção**. Para ficar mais claro vamos ao código:

```
class Cor:  
    vermelho = 1  
    verde = 2  
    azul = 3  
    branco = 4  
    preto = 5  
  
# Mude a cor para testar  
cor_atual = 2  
  
if cor_atual == Cor.vermelho:  
    print("Vermelho")  
elif cor_atual == Cor.verde:  
    print("Verde")  
elif cor_atual == Cor.azul:  
    print("Azul")  
elif cor_atual == Cor.branco:  
    print("Branco")  
elif cor_atual == Cor.preto:  
    print("Preto")  
else:  
    print("Desconhecido")
```

Representação visual:

The diagram illustrates a memory model for a variable assignment. On the left, a purple box labeled "Global frame" contains two entries: "Cor" with value 2 and "cor_atual" with value 2. A blue arrow points from the "Cor" entry to a table on the right. The table is titled "Cor class" and has a sub-section "hide attributes". It lists five entries: "azul" (value 3), "branco" (value 4), "preto" (value 5), "verde" (value 2), and "vermelho" (value 1). The "vermelho" entry is highlighted with a red border.

Cor class	
hide attributes	
azul	3
branco	4
preto	5
verde	2
vermelho	1

11 - Estruturas de repetição

While

Python tem dois tipos de comandos para executar comandos em loop (executar repetidas vezes uma instrução) o **while** e o **for**, inicialmente vamos entender como funciona o **while**. While do inglês, enquanto, ou seja, enquanto uma determinada condição for válida, a ação continuará sendo repetida. Por exemplo:

```
a = 1  
  
while a < 8:  
    print(a)  
    a += 1
```

Declarada a variável **a**, de valor inicial **1** (pode ser qualquer valor, inclusive zero) colocamos a condição de que, enquanto o valor de **a** for menor que **8**, imprime o valor de **a** e acrescente (some) **1**, repetidamente.

O retorno será:

```
2  
3  
4  
5  
6  
7
```

Repare que isto é um loop, ou seja, a cada ação o bloco de código salva seu último estado e repete a instrução, até atingir a condição proposta.

Outra possibilidade é de que durante uma execução de **while**, podemos programar um **break** (comando que para a execução de um determinado bloco de código ou instrução) que acontece se determinada condição for atingida. Normalmente o uso de **break** se dá quando colocamos mais de uma condição que, se a instrução do código atingir qualquer uma dessas condições (uma delas) ele para sua execução para que não entre em um loop infinito de repetições. Por exemplo:

```
a = 1  
  
while a < 10:  
    print(a)  
    a += 1  
    if a == 4:  
        break
```

Enquanto a variável **a** for menor que **10**, continue imprimindo ela e acrescentando **1** ao seu valor. Mas se em algum momento ela for igual a **4**, pare a repetição. Como explicado anteriormente, existem duas condições, se a execução do código chegar em uma delas, ele já dá por encerrada sua execução. Neste caso, se em algum momento o valor de **a** for **4** ou for um número maior que **10** ele para sua execução.

O resultado será:

2
3
4

For

O comando **for** será muito utilizado quando quisermos trabalhar com um laço de repetição onde conhecemos os seus limites, ou seja, quando temos um objeto em forma de lista ou dicionário e queremos que uma variável percorra cada elemento dessa lista/dicionário interagindo com o mesmo. Ex:

```
compras = ['Arroz', 'Feijão', 'Carne', 'Pão']
```

```
for i in compras:  
    print(i)
```

O retorno será: **Arroz**

Feijão

Carne

Pão

Note que inicialmente declaramos uma variável em forma de lista de nome **compras**, ele recebe 4 elementos do tipo **string** em sua composição. Em seguida declaramos o laço **for** e uma variável temporária de nome **i** (você pode usar o nome que quiser, e essa será uma variável temporária, apenas instanciada nesse laço / nesse bloco de código) que para cada execução dentro de **compras**, irá imprimir o próprio valor. Em outras palavras, a primeira vez que **i** entra nessa lista ela faz a leitura do elemento indexado na posição **0** e o imprime, encerrado o laço essa variável **i** agora entra novamente nessa lista e faz a leitura e exibição do elemento da posição **1** e assim por diante, até o último elemento encontrado nessa lista.

Outro uso bastante comum do **for** é quando sabemos o tamanho de um determinado intervalo, o número de elementos de uma lista, etc... e usamos seu método **in range** para que seja explorado todo esse intervalo. Ex:

```
for x in range(0, 6):  
    print(f'Número {x}')
```

Repare que já de início existe o comando **for**, seguido de uma variável temporária **x**, logo em seguida está o comando **in range**, que basicamente define um intervalo a percorrer (de 0 até 6). Por fim há o comando **print** sobre a variável **x**.

O retorno será: **Número 0**

Número 1
Número 2
Número 3
Número 4
Número 5

*Note que a contagem dos elementos deste intervalo foi de **1** a **5**, **6** já está fora do range, serve apenas como orientação para o interpretador de que ali é o fim deste intervalo. Em Python não é feita a leitura deste último dígito indexado, o interpretador irá identificar que o limite máximo desse intervalo é 6, sendo **5** seu último elemento.

Quando estamos trabalhando com um intervalo há a possibilidade de declararmos apenas um valor como parâmetro, o interpretador o usará como orientação para o fim de um intervalo. Ex:

```
for x in range(6):
    print(f'Número {x}')
```

O retorno será: **Número 0**

Número 1
Número 2
Número 3
Número 4
Número 5

Outro exemplo comum é quando já temos uma lista de elementos e queremos a percorrer e exibir seu conteúdo.

```
lista = ['Pedro', 255, 'Letícia']

for n in nomes:
    print(n)
```

Repare que existe uma lista inicial, com 3 dados inclusos, já quando executamos o comando **for** ele internamente irá percorrer todos valores contidos na lista nomes e incluir os mesmos na variável **n**, independentemente do tipo de dado que cada elemento da lista é, por fim o comando **print** foi dado em cima da variável **n** para que seja exibido ao usuário cada elemento dessa lista.

O resultado será: **Pedro**

255
Letícia

Em Python o laço **for** pode nativamente trabalhar como uma espécie de condicional, sendo assim podemos usar o comando **else** para incluir novas instruções no mesmo. Ex:

```
nomes = ['Pedro', 'João', 'Letícia']

for laco in nomes:
    print(laco)
```

```
else:  
    print('---Fim da lista!!!---')
```

O resultado será: **Pedro**
João
Leticia
---Fim da lista!!!---



Por fim, importante salientar que como for serve como laço de repetição ele suporta operações dentro de si, desde que essas operações requeiram repetidas instruções obviamente. Por exemplo:

```
for x in range(11):  
    for y in range(11):  
        print(f'{x} x {y} = {x * y}')
```

O retorno será toda a tabuada de **0 x 0** até **10 x 10** (que por convenção não irei colocar toda aqui obviamente...).

```
"C:\Users\Fernando\PycharmProjects\Exercícios"  
0 x 0 = 0 1 x 0 = 0 2 x 0 = 0  
0 x 1 = 0 1 x 1 = 1 2 x 1 = 2  
0 x 2 = 0 1 x 2 = 2 2 x 2 = 4  
0 x 3 = 0 1 x 3 = 3 2 x 3 = 6  
0 x 4 = 0 1 x 4 = 4 2 x 4 = 8  
0 x 5 = 0 1 x 5 = 5 2 x 5 = 10  
0 x 6 = 0 1 x 6 = 6 2 x 6 = 12  
0 x 7 = 0 1 x 7 = 7 2 x 7 = 14  
0 x 8 = 0 1 x 8 = 8 2 x 8 = 16  
0 x 9 = 0 1 x 9 = 9 2 x 9 = 18  
0 x 10 = 0 1 x 10 = 10 2 x 10 = 20 etc...
```


12 – Strings

Como já vimos algumas vezes ao longo da apostila, um objeto/variável é um espaço alocado na memória onde armazenaremos um tipo de dado ou informação para que o interpretador trabalhe com esses dados na execução do programa.

Uma **string** é o tipo de dado que usamos quando queremos trabalhar com qualquer tipo de texto, ou conjunto ordenado de caracteres alfanuméricos em geral. Quando atribuímos um conjunto de caracteres, representando uma palavra/texto, devemos obrigatoriamente seguir a sintaxe correta para que o interpretador leia os dados como tal.

A sintaxe para qualquer tipo de texto é basicamente colocar o conteúdo desse objeto entre aspas '', uma vez atribuído dados do tipo **string** para uma variável, por exemplo **nome = 'Maria'**, devemos lembrar de o referenciar como tal. Por exemplo em uma máscara lembrar de que o tipo de dado é **%s**.

Trabalhando com strings

No Python 3, se condicionou usar por padrão '' aspas simples para que se determine que aquele conteúdo é uma **string**, porém em nossa língua existem expressões que usam ' como apóstrofe, isso gera um erro de sintaxe ao interpretador. Por exemplo:

```
'marca d'água'
```

O retorno será um erro de sintaxe porque o interpretador quando abre aspas ele espera que feche aspas apenas uma vez, nessa expressão existem 3 aspas, o interpretador fica esperando que você "feche" aspas novamente, como isso não ocorre ele gera um erro.

O legal é que é muito fácil contornar uma situação dessas, uma vez que python suporta, com a mesma função, " "aspas duplas, usando o mesmo exemplo anterior, se você escrever "marca d'água" ele irá ler perfeitamente todos caracteres (incluindo a apóstrofe) como string. O mesmo ocorre se invertermos a ordem de uso das aspas. Por exemplo:

```
frase1 = 'Era um dia "muuuito" frio'
```

```
print(frase1)
```

O retorno será: **Era um dia "muuuito" frio**

```
Global frame  
frase1 | "Era um dia \"muuuito\" frio"
```

Vimos anteriormente, na seção de comentários, que uma forma de comentar o código, quando precisamos fazer um comentário de múltiplas linhas, era as colocando entre """ aspas triplas (pode ser "" aspas simples ou """ aspas duplas). Um texto entre aspas triplas é interpretado pelo interpretador como um comentário, ou seja, o seu conteúdo será por padrão ignorado, mas se atribuirmos esse texto de várias linhas a uma variável, ele passa a ser um objeto comum, do tipo **string**.

```
"""Hoje tenho treino  
Amanhã tenho aula  
Quinta tenho consulta"""
```

*Esse texto nessa forma é apenas um comentário.

```
texto1 = """Hoje tenho treino  
Amanhã tenho aula
```

```
Quinta tenho consulta"""
```

```
print(texto1)
```

*Agora esse texto é legível ao interpretador, inclusive você pode printar ele ou usar da forma como quiser, uma vez que agora ele é uma **string**.

```
Global frame
texto1 "Hoje tenho treino
          Amanhã tenho aula
          Quinta tenho consulta"
```

Formatando uma string

Quando estamos trabalhando com uma **string** também é bastante comum que por algum motivo precisamos formatá-la de alguma forma, e isso também é facilmente realizado desde que dados os comandos corretos.

Convertendo uma string para minúsculo

```
frase1 = 'A linguagem Python é muito fácil de aprender.'
```

```
print(frase1.lower())
```

Repare que na segunda linha o comando `print()` recebe como parâmetro o conteúdo da variável `frase1` acrescido do comando `.lower()`, convertendo a exibição dessa **string** para todos caracteres minúsculos.

O retorno será: **a linguagem python é muito fácil de aprender.**

Convertendo uma string para maiúsculo

Da mesma forma o comando `.upper()` fará o oposto de `.lower()`, convertendo tudo para maiúsculo. Ex:

```
frase1 = 'A linguagem Python é muito fácil de aprender.'
```

```
print(frase1.upper())
```

O retorno será: **A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.**

Lembrando que isso é uma formatação, ou seja, os valores da **string** não serão modificados, essa modificação não é permanente, ela é apenas a maneira com que você está pedindo para que o conteúdo da **string** seja mostrado.

Se você usar esses comandos em cima da variável em questão aí sim a mudança será permanente, por exemplo:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
```

```
frase1 = frase1.lower()
```

```
print(frase1)
```

O retorno será: **a linguagem python é muito fácil de aprender.**

Nesse caso você estará alterando permanentemente todos caracteres da **string** para minúsculo.

Buscando dados dentro de uma string

Você pode buscar um dado dentro do texto convertendo ele para facilitar achar esses dados, por exemplo um nome que você não sabe se lá dentro está escrito com a inicial maiúscula, todo em maiúsculo ou todo em minúsculo, para não ter que testar as 3 possibilidades, você pode usar comandos como:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'  
frase1 = frase1.lower()  
  
print('py' in frase1.lower())
```

O retorno será **True**. Primeiro toda **string** foi convertida para minúscula, por segundo foi procurado '**py**' que nesse caso consta dentro da **string**.

Desmembrando uma string

Outro comando interessante é o `.split()`, ele irá desmembrar uma **string** em palavras separadas, para que você possa fazer a formatação ou o uso de apenas uma delas, por exemplo:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'  
frase1 = frase1.lower()  
  
print(frase1.split())
```

O resultado será: `['a', 'linguagem', 'python', 'é', 'muito', 'fácil', 'de', 'aprender.]`

Alterando a cor de um texto

Supondo que você quer imprimir uma mensagem de erro que chame a atenção do usuário, você pode fazer isso associando também uma cor a este texto, por meio de um código de cores, é possível criar variáveis que irão determinar a cor de um texto, por exemplo:

```
ERRO = '\033[91m' #código de cores "vermelho"
```

```
NORMAL = '\033[0m'
```

```
print(ERRO, 'Mensagem de erro aqui', NORMAL)
```

O Resultado será: **Mensagem de erro aqui.**

Uma vez que você declarou as cores como variáveis, você pode incorporar elas nos próprios parâmetros do código (note também que aqui como é uma variável criada para um caso especial, ela inclusive foi criada com nomenclatura toda maiúscula para eventualmente destaca-la de outras variáveis comuns no corpo do código).

```
print(ERRO + 'Mensagem de erro' + NORMAL)
```

O resultado será: **Mensagem de erro aqui.**

Apenas como referência, segue uma tabela com os principais códigos ANSI de cores tanto para a fonte quanto para o fundo.

Cor	Fonte	Fundo
Preto	\033[1;30m	\033[1;40m
Vermelho	\033[1;31m	\033[1;41m
Verde	\033[1;32m	\033[1;42m
Amarelo	\033[1;33m	\033[1;43m
Azul	\033[1;34m	\033[1;44m
Magenta	\033[1;35m	\033[1;45m
Cyan	\033[1;36m	\033[1;46m
Cinza Claro	\033[1;37m	\033[1;47m
Cinza Escuro	\033[1;90m	\033[1;100m
Vermelho Claro	\033[1;91m	\033[1;101m
Claro		
Verde Claro	\033[1;92m	\033[1;102m
Amarelo Claro	\033[1;93m	\033[1;103m
Claro		
Azul Claro	\033[1;94m	\033[1;104m
Magenta Claro	\033[1;95m	\033[1;105m
Claro		
Cyan Claro	\033[1;96m	\033[1;106m

Branco

\033[1;97m \033[1;107m

Alterando a posição de exibição de um texto

Estamos acostumados a, enquanto trabalhamos dentro de um editor de texto, usar do artifício de botões de atalho que podem fazer com que um determinado texto fique centralizado ou alinhado a um dos lados da página, por exemplo. Internamente isto é feito pelos comandos **center()** (centralizar), **ljust()** (alinhar à esquerda) e **rjust()** (alinhar à direita).

Exemplo de centralização:

```
frase1 = 'Bem Vindo ao Meu Programa!!!'
```

```
print(frase1.center(50))
```

Repare que a função **print** aqui tem como parâmetro a variável **frase1** acrescida do comando **center(50)**, ou seja, centralizar dentro do intervalo de 50 caracteres. (A **string** inteira terá 50 caracteres, como **frase1** é menor do que isto, os outros caracteres antes e depois dela serão substituídos por espaços.

O retorno será: ‘ Bem Vindo ao Meu Programa!!! ’

Exemplo de alinhamento à direita:

```
frase1 = 'Bem Vindo ao Meu Programa!!!'
```

```
print(frase1.rjust(50))
```

O retorno será: ‘ Bem Vindo ao Meu Programa!!! ’

Formatando a apresentação de números em uma string

Existirão situações onde, seja em uma string ou num contexto geral, quando pedirmos ao Python o resultado de uma operação numérica ou um valor pré estabelecido na matemática (como pi por exemplo) ele irá exibir este número com mais casas decimais do que o necessário. Ex:

```
from math import pi  
  
print('O número pi é: {:.2f}')
```

Na primeira linha estamos importando o valor de **pi** da biblioteca externa **math**. Em seguida dentro da **string** estamos usando uma máscara que irá ser substituída pelo valor de **pi**, mas nesse caso, como padrão.

O retorno será: **O número pi é: 3.141592653589793**

Num outro cenário, vamos imaginar que temos uma variável com um valor int extenso, mas só queremos exibir suas duas primeiras casas decimais, nesse caso, isto pode ser feito pelo comando **.f** e uma máscara simples. Ex:

```
num1 = 34.295927957329247  
  
print('O valor da ação fechou em {:.2f}'.format(num1))
```

Repare que dentro da **string** existe uma máscara de substituição **%** seguida de **.2f**, estes 2f significam ao interpretador que nós queremos que sejam exibidas apenas duas casas decimais após a vírgula. Neste caso o retorno será: **O valor da ação fechou em 34.30**

Usando o mesmo exemplo, mas substituindo **.2f** por **.5f**, o resultado será: **O valor da ação fechou em 34.29593** (foram exibidas 5 casas “após a vírgula”).

13 – Listas

Listas em Python são o equivalente a **Arrays** em outras linguagens de programação, mas calma que se você está começando do zero, e começando com Python, esse tipo de conceito é bastante simples de entender.

Listas são um dos tipos de dados aos quais iremos trabalhar com frequência, uma lista será um objeto que permite guardar diversos dados dentro dele, de forma organizada e indexada. É como se você pegasse várias variáveis de vários tipos e colocasse em um espaço só da memória do seu computador, dessa maneira, com a indexação correta, o interpretador consegue buscar e ler esses dados de forma muito mais rápida do que trabalhar com eles individualmente. Ex:

```
lista = []
```

Podemos facilmente criar uma lista com já valores inseridos ou adiciona-los manualmente conforme nossa necessidade, sempre respeitando a sintaxe do tipo de dado. Ex:

```
lista2 = [1, 5, 'Maria', 'João']
```

Aqui criamos uma lista já com 4 dados inseridos no seu índice, repare que os tipos de dados podem ser mesclados, temos **ints** e **strings** na composição dessa lista, sem problema algum. Podemos assim deduzir também que uma lista é um tipo de variável onde conseguimos colocar diversos tipos de dados sem causar conflito.



Adicionando dados manualmente

Se queremos adicionar manualmente um dado ou um valor a uma lista, este pode facilmente ser feito pelo comando `append()`. Ex:

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)

print(lista2)
```

Irá adicionar o valor **4** na posição zero do índice da lista.

O retorno será: **[1, 5, 'Maria', 'João', 4]**



Lembrando que por enquanto, esses comandos são sequenciais, ou seja, executar o primeiro `.append` irá colocar um dado armazenado na posição 0 do índice, o segundo `.append` automaticamente irá guardar um dado na posição 1 do índice, e assim por diante...

Removendo dados manualmente

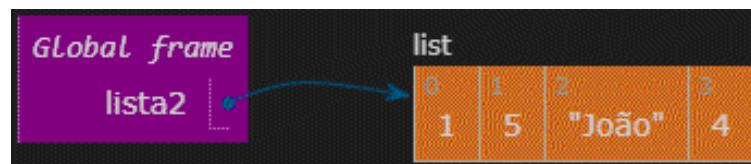
Da mesma forma, podemos executar o comando `.remove()` para remover o conteúdo de algum índice da lista.

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)
lista2.remove('Maria')

print(lista2)
```

Irá remover o dado **Maria**, que nesse caso estava armazenado na posição **2** do índice.

O retorno será: **[1, 5, 'João', 4]**



Lembrando que no comando `.remove` você estará dizendo que conteúdo você quer excluir da lista, supondo que fosse uma lista de nomes `['Paulo', 'Ana', 'Maria']` o comando `.remove('Maria')` irá excluir o dado **'Maria'**, o próprio comando busca dentro da lista onde esse dado específico estava guardado e o remove, independentemente de sua posição.

Removendo dados via índice

Para deletarmos o conteúdo de um índice específico, basta executar o comando **del lista[nº do índice]**

```
lista2 = ['Ana', 'Carlos', 'João', 'Sonia']
del lista2[2]

print(lista2)
```

O retorno será: ['Ana', 'Carlos', 'Sonia']



Repare que inicialmente temos uma lista ['Ana', 'Carlos', 'João', 'Sonia'] e executarmos o comando **del lista[2]** iremos deletar 'João' pois ele está no índice 2 da lista. Nesse caso 'Sonia' que era índice 3 passa a ser índice 2, ele assume a casa anterior, pois não existe “espaço vazio” numa lista.

Verificando a posição de um elemento

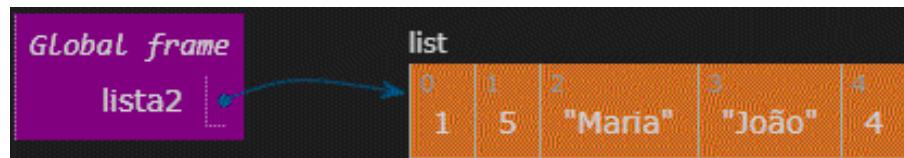
Para verificarmos em que posição da lista está um determinado elemento, basta executarmos o comando `.index()` Ex:

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)

print(lista2.index('Maria'))
```

O retorno será **2**, porque 'Maria' está guardado no índice **2** da lista.

Representação visual:



Se você perguntar sobre um elemento que não está na lista o interpretador simplesmente retornará uma mensagem de erro dizendo que de fato, aquele elemento não está na nossa lista.

Verificando se um elemento consta na lista

Podemos também trabalhar com operadores aqui para consultar em nossa lista se um determinado elemento consta nela ou não. Por exemplo, executando o código '**'João' in lista**' devemos receber um retorno True ou False.

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)

print('João' in lista2)
```

O retorno será: **True**

Formatando dados de uma lista

Assim como usamos comandos para modificar/formatar uma **string** anteriormente, podemos aplicar os mesmos comandos a uma lista, porém é importante que fique bem claro que aqui as modificações ficam imediatamente alteradas (no caso da **string**, nossa formatação apenas alterava a forma como seria exibida, mas na variável a **string** permanecia intacta). Ex:

```
lista1 = []
lista1.append('Paulo')
lista1.append('Maria')

print(lista1)
```

Inicialmente criamos uma lista vazia e por meio do método **append()** inserimos nela duas **strings**. Por meio da função **print()** podemos exibir em tela seu conteúdo:

O retorno será **[Paulo, Maria]**

Representação visual:



```
lista1.reverse()
```

```
print(lista1)
```

Se fizermos o comando **lista1.reverse()** e em seguida dermos o comando **print(lista1)** o retorno será **[Maria, Paulo]**, e desta vez esta alteração será permanente.

Listas dentro de listas

Também é possível adicionar listas dentro de listas, parece confuso, mas na prática, ao inserir um novo dado dentro de um índice, se você usar a sintaxe de lista [] você estará adicionando, ali naquela posição do índice, uma nova lista. Ex:

```
lista = [1, 2, 4, 'Paulo']
```

Adicionando uma nova lista no índice 4 da lista atual ficaria:

```
lista = [1, 2, 4, 'Paulo']
```

```
lista = [1, 2, 4, 'Paulo', [2, 5, 'Ana']]
```

```
print(lista)
```

O retorno será [1, 2, 4, 'Paulo', [2, 5, 'Ana']]

Representação visual:



Tuplas

Uma **Tupla** trabalha assim como uma lista para o interpretador, mas a principal diferença entre elas é que lista é dinâmica (você pode alterá-la à vontade) enquanto uma tupla é estática (elementos atribuídos são fixos) e haverão casos onde será interessante usar uma ou outra.

Um dos principais motivos para se usar uma tupla é o fato de poder incluir um elemento, o nome Paulo por exemplo, várias vezes em várias posições do índice, coisa que não é permitida em uma lista, a partir do momento que uma lista tem um dado ou valor atribuído, ele não pode se repetir.

Segunda diferença é que pela sintaxe uma lista é criada a partir de colchetes, uma tupla a partir de parênteses novamente.

```
minhatupla = tuple()
```

Para que o interpretador não se confunda, achando que é um simples objeto com um parâmetro atribuído, pela sintaxe, mesmo que haja só um elemento na tupla, deve haver ao menos uma vírgula como separador. Ex:

```
minhatupla = tuple(1,
```

Se você executar **type(minhatupla)** você verá **tuple** como retorno, o que está correto, se não houvesse a vírgula o interpretador iria retornar o valor **int** (já que 1 é um número inteiro).

Se você der um comando **dir(minhatupla)** você verá que basicamente temos **index** e **count** como comandos padrão que podemos executar, ou seja, ver seus dados e contá-los também.

Você pode acessar o índice exatamente como fazia com sua lista. Ex:

```
minhatupla = tuple(1,)  
minhatupla[0]
```

O retorno será: **1**

Outro ponto a ser comentado é que, se sua tupla possuir apenas um elemento dentro de si, é necessário indicar que ela é uma tupla ou dentro de si colocar uma vírgula como separador mesmo que não haja um segundo elemento. Já quando temos 2 ou mais elementos dentro da tupla, não é mais necessário indicar que ela é uma tupla, o interpretador identificará automaticamente. Ex:

```
minhatupla = tuple('Ana')  
minhatupla2 = ('Ana',)  
minhatupla3 = ('Ana', 'Maria')
```

Em **minhatupla** existe a atribuição de tipo **tuple()**, em **minhatupla2** existe a vírgula como separador mesmo não havendo um segundo elemento, em **minhatupla3** já não é mais necessário isto para que o identifique o tipo de dado como tupla.

Como uma tupla tem valores já predefinidos e imutáveis, é comum haver mais do mesmo elemento em diferentes posições do índice. Ex:

```
tupladores = ('azul', 'branco', 'azul', 'vermelho', 'preto', 'azul', 'amarelo')
```

Executando o código **tupladores.count('azul')**

```
tupladores = ('azul', 'branco', 'azul', 'vermelho', 'preto', 'azul', 'amarelo')
```

```
print(tupladores.count('azul'))
```

O retorno será **3** porque existem dentro da tupla 'azul' 3 vezes.

Representação visual:



Pilhas

Pilhas nada mais são do que listas em que a inserção e a remoção de elementos acontecem na mesma extremidade. Para abstrairmos e entendermos de forma mais fácil, imagine uma pilha de papéis, se você colocar mais um papel na pilha, será em cima dos outros, da mesma forma que o primeiro papel a ser removido da pilha será o do topo.

Adicionando um elemento ao topo de pilha

Para adicionarmos um elemento ao topo da pilha podemos usar o nosso já conhecido `.append()` recebendo o novo elemento como parâmetro.

```
pilha = [10, 20, 30]  
pilha.append(50)  
  
print(pilha)
```

O retorno será: **[10, 20, 30, 50]**

Removendo um elemento do topo da pilha

Para removermos um elemento do topo de uma pilha podemos usar o comando `.pop()`, e neste caso como está subentendido que será removido o último elemento da pilha (o do topo) não é necessário declarar o mesmo como parâmetro.

```
pilha = [10, 20, 30]  
pilha.pop()
```

```
print(pilha)
```

O retorno será: **[10, 20]**

Consultando o tamanho da pilha

Da mesma forma como consultamos o tamanho de uma lista, podemos consultar o tamanho de uma pilha, para termos noção de quantos elementos ela possui e de qual é o topo da pilha. Ex:

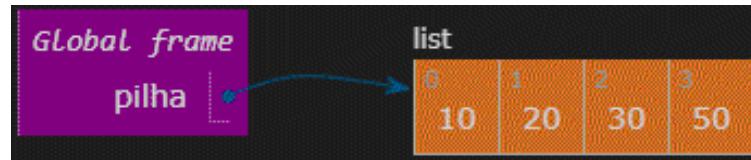
```
pilha = [10, 20, 30]  
pilha.append(50)
```

```
print(pilha)  
print(len(pilha))
```

O retorno será: [10, 20, 30, 50]

4

Representação visual:



Na primeira linha, referente ao primeiro **print()**, nos mostra os elementos da pilha (já com a adição do “50” em seu topo. Por fim na segunda linha, referente ao segundo **print()** temos o valor 4, dizendo que esta pilha tem 4 elementos;

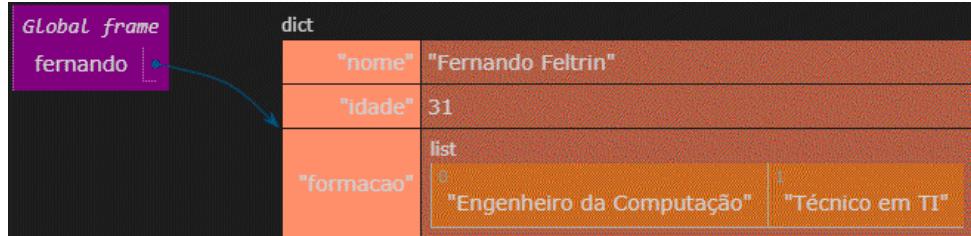
14 – Dicionários

Enquanto listas e tuplas são estruturas indexadas, um dicionário, além da sintaxe também diferente, se resume em **chave:valor**. Assim como em um dicionário normal você tem uma palavra e seu significado, aqui a estrutura lógica será essa.

A sintaxe de um dicionário é definida por chaves {}, deve seguir a ordem **chave:valor** e usar vírgula como separador, para não gerar um erro de sintaxe. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31, 'formacao': ['Engenheiro da Computação', 'Técnico em TI']}
```

Repare que inicialmente o dicionário foi atribuído a um objeto, depois que seguindo a sintaxe foram criados 3 campos (nome, idade e formação) e dentro de formação ainda foi criado uma lista, onde foram adicionados dois elementos ao índice.



Uma prática comum tanto em listas, quanto em dicionários, é usar de uma tabulação que torne visualmente o código mais organizado, sempre que estivermos trabalhando com estes tipos de dados teremos uma vírgula como separador dos elementos, e é perfeitamente normal após uma vírgula fazer uma quebra de linha para que o código fique mais legível. Esta prática não afeta em nada a leitura léxica do interpretador nem a performance de operação do código.

```
fernando = {'nome': 'Fernando Feltrin',  
           'idade': 31,  
           'formacao': ['Engenheiro da Computação',  
                        'Técnico em TI']}
```

Executando um **type(fernando)** o retorno será **dict**, porque o interpretador, tudo o que estiver dentro de {} chaves ele interpretará como chaves e valores de um dicionário.

Assim como existem listas dentro de listas, você pode criar listas dentro de dicionários e até mesmo dicionários dentro de dicionários sem problema algum.

Você pode usar qualquer tipo de dado como chave e valor, o que não pode acontecer é esquecer de declarar um ou outro pois irá gerar erro de sintaxe.

Da mesma forma como fazíamos com listas, é interessante saber o tamanho de um dicionário, e assim como em listas, o comando `len()` agora nos retornará à quantidade de **chaves:valores** inclusos no dicionário. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,  
            'formacao': ['Engenheiro da Computação', 'Téc em TI']}
```



```
print(len(fernando))
```

O retorno será **3**.

Consultando chaves/valores de um dicionário

Você pode consultar o valor de dentro de um dicionário pelo comando `.get()`. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,  
            'formacao': ['Engenheiro da Computação', 'Técnico em TI']}
```



```
print(fernando.get('idade'))
```

O retorno será **31**.

Consultando as chaves de um dicionário

É possível consultar quantas e quais são as chaves que estão inclusas dentro de um dicionário pelo comando `.keys()`. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,  
            'formacao': ['Engenheiro da Computação', 'Técnico em TI']}
```

```
print(fernando.keys())
```

O retorno será: `dict_keys(['nome', 'idade', 'formacao'])`

Consultando os valores de um dicionário

Assim como é possível fazer a leitura somente dos valores, pelo comando `.values()`.
Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,  
            'formacao': ['Engenheiro da Computação', 'Técnico em TI']}
```



```
print(fernando.values())
```

O retorno será: `dict_values(['Fernando Feltrin', 31, ['Engenheiro da Computação', 'Técnico em TI']])`

Mostrando todas chaves e valores de um dicionário

Por fim o comando `.items()` retornará todas as chaves e valores para sua consulta. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,  
            'formacao': ['Engenheiro da Computação', 'Técnico em TI']}  
  
print(fernando.items())
```

O retorno será: `dict_items([('nome': 'Fernando Feltrin', 'idade': 31, 'formacao': ['Engenheiro da Computação', 'Técnico em TI'])])`

Manipulando dados de um dicionário

Quanto temos um dicionário já com valores definidos, pré-programados, mas queremos alterá-los, digamos, atualizar os dados dentro de nosso dicionário, podemos fazer isso manualmente através de alguns simples comandos.

Digamos que tínhamos um dicionário inicial:

```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}
```



```
print(pessoa)
```

O retorno será: `{'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'], 'nacionalidade': 'brasileiro'}`

Representação visual:



```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}
```

```
pessoa['idade'] = 44
```

```
print(pessoa)
```

Executando o comando `pessoa['idade'] = 44` estaremos atualizando o valor `'idade'` para `44` no nosso dicionário, consultando o dicionário novamente você verá que a idade foi atualizada.

Executando novamente `print(pessoa)` o retorno será: `{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia'], 'nacionalidade': 'brasileiro'}`

Adicionando novos dados a um dicionário

Além de substituir um valor por um mais atual, também é possível adicionar manualmente mais dados ao dicionário, assim como fizemos anteriormente com nossas listas, através do comando `.append()`. Ex:

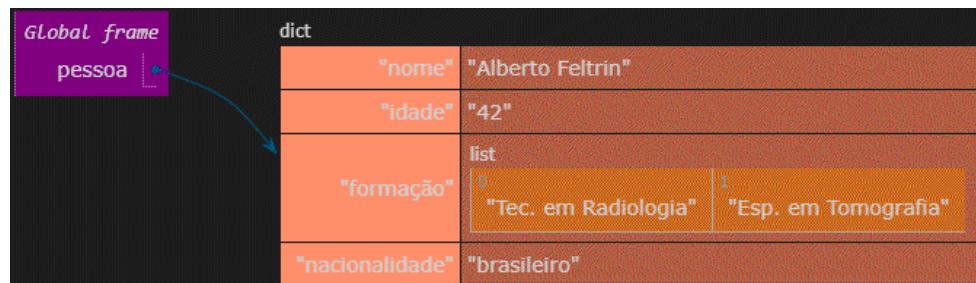
```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}

pessoa['formação'].append('Esp. em Tomografia')

print(pessoa)
```

O retorno será: `{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia', 'Esp em Tomografia'], 'nacionalidade': 'brasileiro'}`

Representação visual:



*Importante salientar que para que você possa adicionar mais valores a uma chave, ela deve respeitar a sintaxe de uma lista.

15 – Conjuntos numéricos

Se você lembrar das aulas do ensino médio certamente lembrará que em alguma etapa lhe foi ensinado sobre conjuntos numéricos, que nada mais era do que uma foram de categorizarmos os números quanto suas características (reais, inteiros, etc...). Na programação isto se repete de forma parecida, uma vez que quando queremos trabalhar com conjuntos numéricos normalmente estamos organizando números.

Um conjunto é ainda outra possibilidade de armazenamento de dados que temos, de forma que ele parece uma lista com sintaxe de dicionário, mas não indexável e que não aceita valores repetidos, confuso não? A questão é que como aqui podemos ter valores repetidos, assim como conjuntos numéricos que usávamos no ensino médio, aqui podemos unir dois conjuntos, fazer a intersecção entre eles, etc... Vamos aos exemplos:

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

Se executarmos o comando **type(a)** o retorno será **set**. O interpretador consegue ver na sintaxe que não é nem uma lista, nem uma tupla e nem um dicionário, mas apenas um conjunto de dados alinhados.

Trabalhando com mais de um conjunto, podemos fazer operações entre eles. Por exemplo:

União de conjuntos

A união de dois conjuntos é feita através do operador `.union()`, de forma que a operação irá na verdade gerar um terceiro conjunto onde constam os valores dos dois anteriores. Ex:

```
c1 = {1, 2}  
c2 = {2, 3}  
c1.union(c2)  
#c1 união com c2, matematicamente juntar 2 conjuntos.
```

O retorno será: **{1, 2, 3}**

Repare que a união dos dois conjuntos foi feita de forma a pegar os valores que não eram comuns aos dois conjuntos e incluir, e os valores que eram comuns aos dois conjuntos simplesmente manter, sem realizar nenhuma operação.

Interseção de conjuntos

Já a interseção entre eles fará o oposto, anulando valores avulsos e mantendo só os que eram comuns aos dois conjuntos. Interseção de dois conjuntos através do operador `.intersection()`. Ex:

```
c1 = {1, 2}  
c2 = {2, 3}  
c1.intersection(c2)
```

O retorno será: `{2}` #Único elemento em comum aos dois conjuntos

Lembrando que quando executamos expressões matemáticas elas apenas alteram o que será mostrado para o usuário, a integridade dos dados iniciais se mantém. Para alterar os dados iniciais, ou seja, salvar essas alterações, é possível executar o comando `.update()` após as expressões, assim os valores serão alterados de forma permanente.
Ex:

```
c1 = {1, 2}  
c2 = {2, 3}  
c1.union(c2)  
c1.update(c2)
```

Agora consultando c1 o retorno será `{1, 2, 3}`

Verificando se um conjunto pertence ao outro

Também é possível verificar se um conjunto está contido dentro do outro, a través da expressão `<=` Ex:

```
c2 <= c1
```

O retorno será **True** porque os valores de **c2** (2 e 3) estão contidos dentro de **c1** (que agora é 1, 2 e 3).

Diferença entre conjuntos

Outra expressão matemática bastante comum é fazer a diferença entre dois conjuntos. Ex:

$\{1, 2, 3\} - \{2\}$

O retorno será: $\{1, 3\}$ #O elemento 2 foi subtraído dos conjuntos.

16 – Interpolação

Quando estamos escrevendo nossos primeiros códigos, costumamos escrever todas expressões de forma literal e, na verdade, não há problema nenhum nisto. Porém quando estamos avançando nos estudos de programação iremos ver aos poucos que é possível fazer a mesma coisa de diferentes formas, e a proficiência em uma linguagem de programação se dá quando conseguirmos criar códigos limpos, legíveis, bem interpretados e enxutos em sua sintaxe.

O ponto que eu quero chegar é que pelas boas práticas de programação, posteriormente usaremos uma simbologia em nossas expressões que deixarão nosso código mais limpo, ao mesmo tempo um menor número de caracteres por linha e um menor número de linhas por código resultam em um programa que é executado com performance superior.

Um dos conceitos que iremos trabalhar é o de máscaras de substituição que como o próprio nome já sugere, se dá a uma série de símbolos reservados pelo sistema que servirão para serem substituídos por um determinado dado ao longo do código.

Vamos ao exemplo:

```
nome1 = input('Digite o seu nome')
```

A partir daí poderíamos simplesmente mandar exibir o nome que o usuário digitou e que foi atribuído a variável **nome1**. Ex:

```
nome1 = input('Digite o seu nome')
```

```
print(nome1)
```

O resultado será o nome que o usuário digitou anteriormente. Porém existe uma forma mais elaborada, onde exibiremos uma mensagem e dentro da mensagem haverá uma máscara que será substituída pelo nome digitado pelo usuário. Vamos supor que o usuário irá digitar “Fernando”, então teremos um código assim:

```
nome1 = input('Digite o seu nome')
```

```
print('Seja bem vindo %s'%(nome1))
```

O resultado será: **Seja bem vindo Fernando**

Repare que agora existe uma mensagem de boas-vindas seguida da máscara **%s** (que neste caso está deixando espaço a ser substituído por uma **string**) e após a frase existe um operador que irá ler o que está atribuído a variável **nome1** e irá substituir pela máscara.

As máscaras de substituição podem suportar qualquer tipo de dado desde que referenciados da forma correta: **%s(string)** ou **%d(número inteiro)** por exemplo.

Também é possível usar mais de uma máscara se necessário, desde que se respeite a ordem das máscaras com seus dados a serem substituídos e a sintaxe equivalente, que agora é representada por { } contendo dentro o número da ordem dos índices dos dados a serem substituídos pelo comando **.format()**. Por exemplo:

```
nota1 = '{0} está reprovado, assim como seu colega {1}'
```

```
print(nota1.format('Pedro', 'Francisco'))
```

O Resultado será: **Pedro está reprovado, assim como seu colega Francisco**

Este tipo de interpolação também funcionará normalmente sem a necessidade de identificar a ordem das máscaras, podendo deixar as { } chaves em branco.

Avançando com interpolações

```
nome = 'Maria'  
idade = 30
```

Se executarmos o comando **print()** da seguinte forma:

```
print('Nome: %s Idade: %d' % (nome, idade))
```

O interpretador na hora de exibir este conteúdo irá substituir **%s** pela variável do tipo **string** e o **%d** pela variável do tipo **int** declarada anteriormente, sendo assim, o retorno será: **Nome: Maria Idade: 30**

Python vem sofrendo uma serie de "atualizações" e uma delas de maior impacto diz respeito a nova forma de se fazer as interpolações, alguns programadores consideraram essa mudança uma "mudança para pior", mas é tudo uma questão de se acostumar com a nova sintaxe.

Por quê estou dizendo isto, porque será bastante comum você pegar exemplos de códigos na web seguindo a sintaxe antiga, assim como códigos já escritos em Python 3 que adotaram essa nova sintaxe que mostrarei abaixo.

Importante salientar que as sintaxes antigas não serão (ao menos por enquanto) descontinuadas, ou seja, você pode escrever usando a que quiser, ambas funcionarão, mas será interessante começar a se acostumar com a nova.

Seguindo o mesmo exemplo anterior, agora em uma sintaxe intermediária:

```
print('Nome: {0} Idade: {1}'.format(nome, idade))
```

O resultado será: **Nome: Maria Idade: 30**

Repare que agora o comando vem seguido de um índice ao qual será substituído pelos valores encontrados no **.format()**.

E agora por fim seguindo o padrão da nova sintaxe, mais simples e funcional que os anteriores:

```
nome = 'Maria'  
idade = 30
```

```
print(f'Nome: {nome}, Idade: {idade}')
```

Repare que o operador **.format()** foi abreviado para simplesmente **f** no início da sentença, e dentro de suas máscaras foram referenciadas as variáveis a serem substituídas.

O resultado será: **Nome: Maria Idade: 30**

O ponto que eu quero que você observe é que à medida que as linguagens de programação vão se modernizando, elas tendem a adotar sintaxes e comandos mais fáceis de serem lidos, codificados e executados.

A nova sintaxe, utilizando de chaves {} para serem interpoladas, foi um grande avanço também no sentido de que entre chaves é possível escrever expressões (por exemplo uma expressão matemática) que ela será lida e interpretada normalmente, por exemplo:

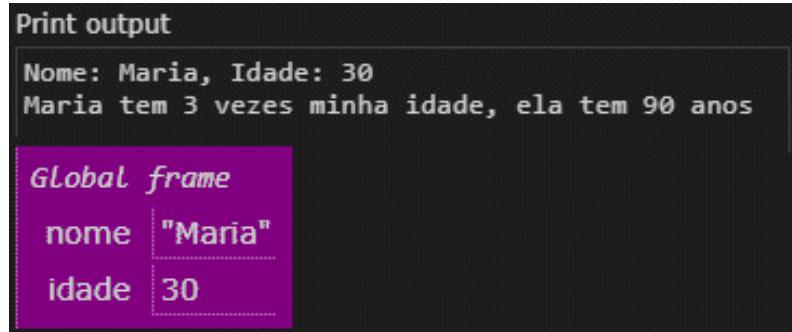
```
nome = 'Maria'  
idade = 30  
  
print(f'Nome: {nome}, Idade: {idade}')  
  
print(f'{nome} tem 3 vezes minha idade, ela tem {3 * idade} anos')
```

O resultado será: **Nome: Maria, Idade: 30**

Maria tem 3 vezes minha idade, ela tem 90 anos

Repare que a expressão matemática de multiplicação foi realizada dentro da máscara de substituição, e ela não gerou nenhum erro pois o interpretador está trabalhando com seu valor atribuído, que neste caso era **30**.

Representação visual:



The screenshot shows a terminal window with two sections. The top section is labeled "Print output" and displays the printed strings: "Nome: Maria, Idade: 30" and "Maria tem 3 vezes minha idade, ela tem 90 anos". The bottom section is labeled "Global frame" and shows a table with two rows: "nome" with value "Maria" and "idade" with value 30.

Global frame	
nome	"Maria"
idade	30

17 – Funções

Já vimos anteriormente as funções `print()` e `input()`, duas das mais utilizadas quando estamos criando estruturas básicas para nossos programas. Agora que você já passou por outros tópicos e já acumula uma certa bagagem de conhecimento, seria interessante nos aprofundarmos mais no assunto funções, uma vez que existem muitas possibilidades novas quando dominarmos o uso das mesmas.

Uma função, independentemente da linguagem de programação, é um bloco de códigos que podem ser executados e reutilizados livremente, quantas vezes forem necessárias. Uma função pode conter dentro de seus parâmetros uma série de instruções a serem executadas cada vez que a referenciamos, sem a necessidade de escrever várias vezes o mesmo bloco de código.

Para ficar mais claro imagine que no corpo de nosso código serão feitas diversas vezes a operação de somar dois valores e atribuir seu resultado a uma variável, é possível criar uma vez esta calculadora de soma e definir ela como uma função, de forma que eu posso posteriormente no mesmo código reutilizar ela apenas mudando seus parâmetros.

Funções predefinidas

A linguagem Python já conta com uma série de funções pré definidas e pré carregadas por sua IDE, prontas para uso. Como dito anteriormente, é muito comum utilizarmos, por exemplo, a função **print()** que é uma função dedicada a exibir em tela o que é lhe dado como parâmetro (o que está dentro dos parênteses). Outro exemplo seria a função **len()** que mostra o tamanho de um determinado dado.

Existe a possibilidade de importarmos módulos externos, ou seja, funções já criadas e testadas que estão disponíveis para o usuário, porém fazem parte de bibliotecas externas, que não vêm carregadas por padrão. Por exemplo é possível importar de uma biblioteca chamada **math** as funções **sin()**, **cos()** e **tg()**, que respectivamente, como seu nome sugere, funções prontas para calcular o seno, o cosseno e a tangente de um determinado valor.

Funções personalizadas

Seguindo o nosso raciocínio, temos funções prontas pré carregadas, funções prontas que simplesmente podemos importar para nosso código e haverão situações onde o meio mais rápido ou prático será criar uma função própria personalizada para uma determinada situação.

Basicamente quando necessitamos criar uma função personalizada, ela começará com a declaração de um **def**, é o meio para que o interpretador assume que a partir dessa palavra reservada está sendo criada uma função. Uma função personalizada pode ou não receber parâmetros (linhas/blocos de código entre parênteses), pode retornar ou não algum dado ou valor ao usuário e por fim, terá um bloco de código indentado para receber suas instruções.

Função simples, sem parâmetros

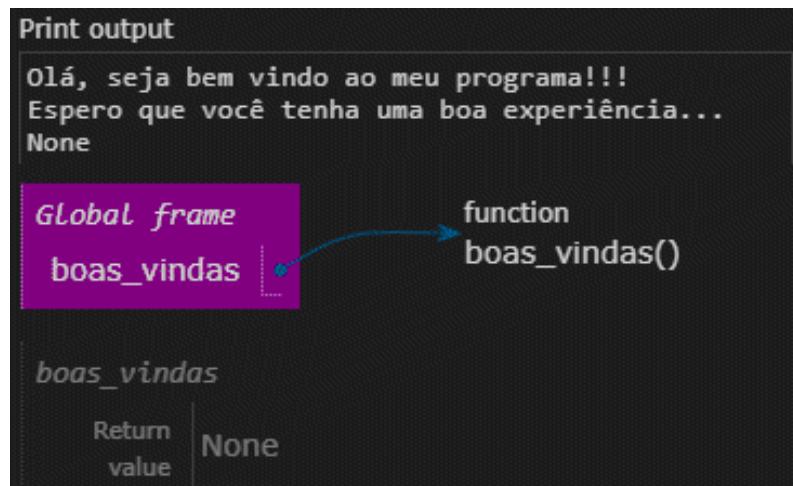
```
def boas_vindas():
    print('Olá, seja bem vindo ao meu programa!!!')
    print('Espero que você tenha uma boa experiência...')

print(boas_vindas())
```

Analizando o código podemos perceber que na primeira linha está o comando **def** seguido do nome dado a nova função, neste caso **boas_vindas()**. Repare que após o nome existe dois pontos : e na linha abaixo, indentado, existem dois comandos **print()** com duas frases. Por fim, dado o comando **print(boas_vindas())** foi chamada a função e seu conteúdo será exibido em tela.

O retorno será: **Olá, seja bem vindo ao meu programa!!!**
Espero que você tenha uma boa experiência...

Representação visual:



Como mencionado anteriormente, uma vez definida essa função **boas_vindas()** toda vez que eu quiser usar ela (seu conteúdo na verdade) basta fazer a referência corretamente.

Função composta, com parâmetros

```
def eleva_numero_ao_cubo(num):
    valor_a_retornar = num * num * num
    return(valor_a_retornar)

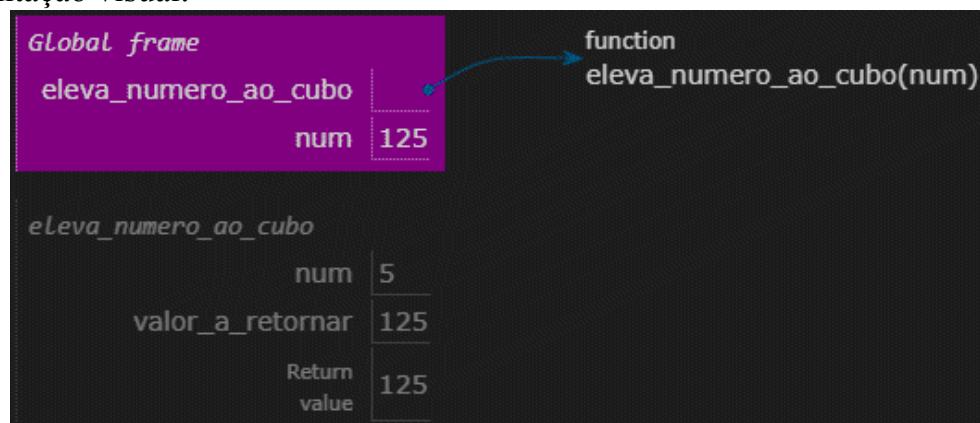
num = eleva_numero_ao_cubo(5)

print(num)
```

Repare que foi definida a função **eleva_numero_ao_cubo(num)** com **num** como parâmetro, posteriormente foi criada uma variável temporária de nome **valor_a_retornar** que pega **num** e o multiplica por ele mesmo 3 vezes (elevando ao cubo), por fim existe a instrução de retornar o valor atribuído a **valor_a_retornar**. Segundo o código existe uma variável **num** que chama a função **eleva_numero_ao_cubo** e dá como parâmetro o valor **5** (que pode ser alterado livremente), e finalizando executa o comando **print()** de **num** que irá executar toda a mecânica criada na função **eleva_numero_ao_cubo** com o parâmetro dado, neste caso **5**.

O retorno será: **125**

Representação visual:



Função composta, com *args e **kwargs

Outra possibilidade que existe em Python é a de trabalharmos com ***args** e com ****kwargs**, que nada mais são do que instruções reservadas ao sistema para que permita o uso de uma quantidade variável de argumentos. Por convenção se usa a nomenclatura **args** e **kwargs** mas na verdade você pode usar o nome que quiser, o marcador que o interpretador buscará na verdade serão os * asteriscos simples ou duplos. Importante entender também que, como estamos falando em passar um número variável de parâmetros, isto significa que internamente no caso de ***args** os mesmos serão tratados como uma lista e no caso de ****kwargs** eles serão tratados como dicionário.

Exemplo *args

```
def print_2_vezes(*args):
    for parametro in args:
        print(parametro + '!' + parametro + '!')

print_2_vezes('Olá Mundo!!! ')
```

Definida a função **print_2_vezes(*args)** é passado como instrução um comando **for**, ou seja um laço que irá verificar se a variável temporária **parametro** está contida em **args**, como instrução do laço é passado o comando **print**, que irá exibir duas vezes e concatenar o que estiver atribuído a variável temporária **parametro**. Por fim, seguindo o código é chamada a função **print_2_vezes** e é passado como parâmetro a string '**Olá Mundo!!!**'. Lembrando que como existem instruções dentro de instruções é importante ficar atento a indentação das mesmas.

O retorno será: **Olá Mundo!!! !Olá Mundo!!!**

Representação visual:



Exemplo de **kwargs:

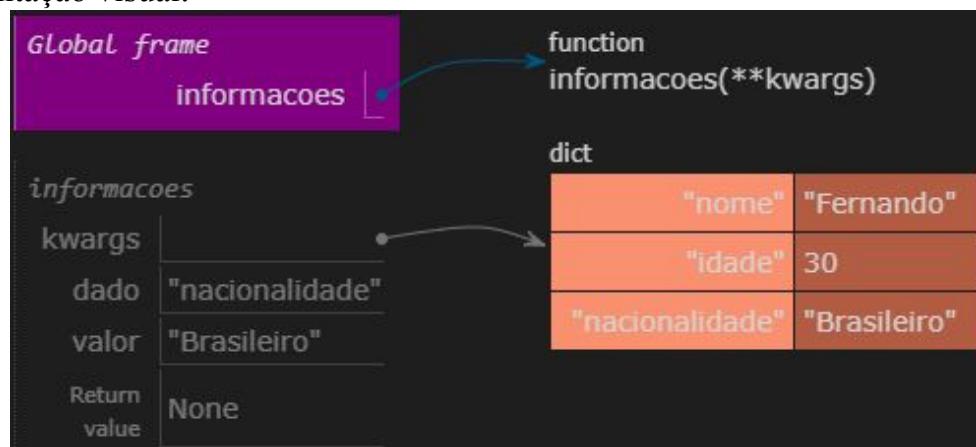
```
def informacoes(**kwargs):
    for dado, valor in kwargs.items():
        print(dado + '-' + str(valor))

informacoes(nome='Fernando', idade=30,
            nacionalidade='Brasileiro')
```

Definida a função **informacoes(**kwargs)** é passado como instrução o comando **for**, que irá verificar a presença dos dados atribuídos a **dado** e **valor**, uma vez que kwargs espera que, como num dicionário, sejam atribuídos chaves:valores. Por fim, se estes parâmetros estiverem contidos em kwargs será dado o comando **print** dos valores de dados e valores, convertidos para **string** e concatenados com o separador -. Então é chamada a função **informacoes** e é passado como chaves e valores os dados contidos no código acima.

O retorno será: **nome-Fernando
idade-30
nacionalidade-Brasileiro**

Representação visual:



18 – Comandos `dir()` e `help()`

O Python como padrão nos fornece uma série de operadores que facilitam as nossas vidas para executar comandos com nossos códigos, mas é interessante você raciocinar que o que convencionalmente usamos quando estamos codificando nossas ideias não chega a ser 10% do que o python realmente tem para oferecer, então, estudando mais a fundo a documentação ou usando o comando `dir`, podemos perguntar a um tipo de objeto (`int`, `float`, `string`, `list`, etc...) todos os recursos internos que ele possui, além disso é possível importar para o python novas funcionalidades, que veremos futuramente neste mesmo curso.

Como exemplo digamos que estamos criando uma lista para guardar nela diversos tipos de dados. Pela sintaxe, uma lista é um objeto/variável que pode receber qualquer nome e tem como característica o uso de colchetes [] para guardar dentro os seus dados. Ex:

```
Lista1 = []
```

Lista vazia

```
Lista2 = [1,'Carlos']
```

Lista com o valor 1 alocado na posição 0 do índice e com 'Carlos' alocado na posição 1 do índice.

Representação visual:



Usando o comando `dir(lista2)` iremos obter uma lista de todas as possíveis funções que podem ser executadas sobre uma lista

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',  
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',  
 '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',  
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',  
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',  
 '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append',  
 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Muito parecido com o `dir()` é o comando `help()`, você pode usar o comando `help(lista2)` e você receberá também uma lista (agora muito mais detalhada) com todas possíveis funções a serem executadas sobre suas listas. Ex:

```
lista2 = [1,'Carlos']
```

```
print(help(lista2))
```

O retorno será:

```
| list(iterable=0, /)
| Built-in mutable sequence.

| If no argument is given, the constructor creates a new empty list.
| The argument must be an iterable if specified.

| Methods defined here:

|_ __add__(self, value, /)
   Return self+value.

|_ __contains__(self, key, /)
   Return key in self.

|_ __delitem__(self, key, /)
   Delete self[key].

|_ __eq__(self, value, /)
   Return self==value.

|_ __ge__(self, value, /)
   Return self>=value.

|_ __getattribute__(self, name, /)
   Return getattr(self, name).

|_ __getitem__(...)
   x.__getitem__(y) <==> x[y]

|_ __gt__(self, value, /)
   Return self>value.

|_ __iadd__(self, value, /)
   Implement self+=value.

|_ __imul__(self, value, /)
   Implement self*=value.

|_ __init__(self, /, *args, **kwargs)
   Initialize self. See help(type(self)) for accurate signature.

|_ __iter__(self, /)
   Implement iter(self).

|_ __le__(self, value, /)
   Return self<=value.

|_ __len__(self, /)
   Return len(self).

|_ __lt__(self, value, /)
   Return self<value.

|_ __mul__(self, value, /)
   Return self**value.
```

```
__ne__(self, value, /)
    Return self!=value.

__repr__(self, /)
    Return repr(self).

__reversed__(self, /)
    Return a reverse iterator over the list.

__rmul__(self, value, /)
    Return value*self.

__setitem__(self, key, value, /)
    Set self[key] to value.

__sizeof__(self, /)
    Return the size of the list in memory, in bytes.

append(self, object, /)
    Append object to the end of the list.

clear(self, /)
    Remove all items from list.

copy(self, /)
    Return a shallow copy of the list.

count(self, value, /)
    Return number of occurrences of value.

extend(self, iterable, /)
    Extend list by appending elements from the iterable.

index(self, value, start=0, stop=9223372036854775807, /)
    Return first index of value.

    Raises ValueError if the value is not present.

insert(self, index, object, /)
    Insert object before index.

pop(self, index=-1, /)
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.

remove(self, value, /)
    Remove first occurrence of value.

    Raises ValueError if the value is not present.

reverse(self, /)
    Reverse *IN PLACE*.

sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.

-----
Static methods defined here:

__new__(*args, **kwargs) from builtins.type
    Create and return a new object. See help(type) for accurate signature.
```

```
| Data and other attributes defined here:
```

```
|     | __hash__ = None
```

Repare que foi classificado e listado cada recurso possível de ser executado em cima de nossa **lista2**, assim como o retorno que é esperado em cada uso.

19 – Builtins e bibliotecas pré-alocadas

Builtins, através do comando `dir()`, nada mais é do que uma forma de você pesquisar quais são os módulos e recursos que já vieram pré-alocados em sua IDE. De forma que você pode pesquisar quais são as funcionalidades que estão disponíveis, e, principalmente as indisponíveis, para que você importe o módulo complementar necessário. Para ficar mais claro, imagine que você consegue fazer qualquer operação matemática básica pois essas funcionalidades já estão carregadas, pré-alocadas e prontas para uso em sua IDE. Caso você necessite usar expressões matemáticas mais complexas, é necessário importar a biblioteca `math` para que novas funcionalidades sejam inclusas em seu código.

Na prática você pode executar o comando `dir(__builtins__)` para ver tudo o que está disponível em sua IDE em tempo real. Ex:

```
print(dir(__builtins__))
```

O retorno será:

```
['ArithError', 'AssertionError', 'AttributeError', 'BaseException',  
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',  
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',  
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',  
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',  
 'FileExistsError', 'FileNotFoundException', 'FloatingPointError', 'FutureWarning',  
 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',  
 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',  
 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError',  
 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',  
 'NotImplementedError', 'OSError', 'OverflowError',  
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',  
 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError',  
 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError',  
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',  
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',  
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',  
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'WindowsError',  
 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__',  
 '__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii',  
 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',  
 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
```

```
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

Cada elemento dessa lista é um módulo pré-alocado, pronta para uso. Importante lembrar que o que faz com que certas palavras e expressões fiquem reservadas ao sistema é justamente um módulo ou biblioteca carregados no builtin. Não é possível excluir tudo o que está pré-carregado, mas só a fim de exemplo, se você excluisse esses módulos básicos o interpretador não teria parâmetros para verificar no código o que é o que de acordo com a sintaxe.

Repare na sintaxe, **__builtins__** é um tipo de variável, mas ela é reservada do sistema, você não pode alterá-la de forma alguma, assim como outros elementos com prefixo ou sufixo **_**.

Importando bibliotecas

Em Python, de acordo com a nossa necessidade, existe a possibilidade de trabalharmos com as bibliotecas básicas já inclusas, ou importarmos outras bibliotecas que nos tragam novas funcionalidades. Python como já dissemos anteriormente algumas vezes, é uma linguagem de programação "com pilhas inclusas", ou seja, as bibliotecas básicas que você necessita para grande parte das funções básicas já vem incluídas e pré-alocadas, de forma que basta chamarmos a função que queremos para de fato usá-la. Porém, dependendo de casos específicos, podemos precisar fazer o uso de funções que não são nativas ou que até mesmo são de bibliotecas externas ao Python padrão, sendo assim necessário que importemos tais bibliotecas para acessarmos suas funções.

Em Python existem duas formas básicas de trabalharmos com as importações, de forma bastante simples, podemos importar uma biblioteca inteira a referenciando pelo nome após o comando **import**, ou podemos importar apenas algo de dentro de uma biblioteca externa para incorporarmos em nosso código, através do comando **from** (nome da biblioteca) **import** (nome da função). Por exemplo:

O Python em todas suas versões já conta com funções matemáticas pré-alocadas, porém para usarmos algumas funções ou constantes matemáticas, é necessário importá-las. Supondo que por algum motivo nosso código precise de **pi**, para que façamos o cálculo de alguma coisa. Podemos definir manualmente o valor de **pi** e atribuir a algum objeto, ou podemos importá-lo de alguma biblioteca externa.

No primeiro exemplo, podemos importar a biblioteca **math**, assim, quando referenciarmos o valor de **pi** ele já estará pré-alocado para uso, por exemplo.

```
import math  
  
raio = 15.3  
  
print('Área do círculo', math.pi * raio ** 2)
```

O valor de **pi**, que faz parte da biblioteca **math** será multiplicado pelo raio e elevado ao quadrado. Como você está importando esse item da biblioteca **math**, não é necessário especificar o seu valor, neste caso, o valor de **pi** (3,1416) já está associado a **pi**.

O resultado será: **735.41**

No segundo exemplo, podemos importar apenas a função **pi** de dentro da biblioteca **math**, da seguinte forma.

```
from math import pi  
  
raio = 15.3  
  
print('Área do círculo', pi * raio ** 2)
```

O valor de **pi** será multiplicado pelo **raio** e elevado ao quadrado.

O retorno será: **735.41**

Por fim, apenas para concluir o nosso raciocínio, uma biblioteca é um conjunto de parâmetros e funções já prontas para facilitar a nossa vida, aqui, podemos definir manualmente o valor de **pi** ou usar ele já pronto de dentro de uma biblioteca.

Apenas faça o seguinte exercício mental, imagine ter que programar manualmente todas as funções de um programa (incluindo as interfaces, entradas e saídas, etc...), seria absolutamente inviável. Todas linguagens de alto nível já possuem suas funções de forma pré-configuradas de forma que basta incorporá-las ao nosso builtin e fazer o uso.

20 – Módulos e pacotes

Um módulo, na linguagem Python, equivale ao método de outras linguagens, ou seja, o programa ele executa dentro de um módulo principal e à medida que vamos o codificando, ele pode ser dividido em partes que podem ser inclusive acessadas remotamente. Um programa bastante simples pode rodar inteiro em um módulo, mas conforme sua complexidade aumenta, e também para facilitar a manutenção do mesmo, ele começa a ser dividido em partes. Já uma função inteira que você escreve e define, e que está pronta, e você permite que ela seja importada e usada dentro de um programa, ela recebe a nomenclatura de um pacote.

Por padrão, implícito, quando você começa a escrever um programa do zero ele está rodando como modulo `__main__`. Quem vem de outras linguagens de programação já está familiarizado a, quando se criava a estrutura básica de um programa, manualmente criar o método main. Em Python essa e outras estruturas básicas já estão pré-definidas e funcionando por padrão, inclusive fazendo a resolução dos nomes de forma interna e automática.

Modularização

Uma vez entendido como trabalhamos com funções, sejam elas pré definidas ou personalizadas, hora de entendermos o que significa modularizações em programação. Apesar do nome assustar um pouco, um módulo nada mais é do que pegarmos blocos de nosso código e o salvar em um arquivo externo, de forma que podemos importar o mesmo posteriormente conforme nossa necessidade. Raciocine que como visto em capítulos anteriores, uma função é um bloco de código que está ali pré-configurado para ser usados dentro daquele código, mas e se pudéssemos salvá-lo de forma a reutilizar o mesmo em outro código, outro programa até mesmo de um terceiro. Isto é possível simplesmente o transformando em um módulo.

Como já mencionado anteriormente diversas vezes, Python é uma linguagem com pilhas inclusas, e isto significa que ela já nos oferece um ambiente pré-configurado com uma série de bibliotecas e módulos pré carregados e prontos para uso. Além disto é possível importar novos módulos para adicionar novas funcionalidades a nosso código assim como também é possível criarmos um determinado bloco de código e o exportar para que se torne um módulo de outro código.

Seguindo com nosso raciocínio aqui é onde começaremos a trabalhar mais diretamente importando e usando o conteúdo disponível em arquivos externos, e isto é muito fácil de se fazer desde que da maneira correta.

Em modularização tudo começa com um determinado bloco de código que se tornará um módulo, código pronto e revisado, sem erros e pronto para executar a sua função desejada, salve o mesmo com um nome de fácil acesso e com a extensão `.py`. Em Python todo arquivo legível pela IDE recebe inicialmente a extensão reservada ao sistema `.py`. Posteriormente compilando o executável de seu programa isto pode ser alterado, mas por hora, os arquivos que estamos trabalhando recebem a extensão `.py`.

Partindo para prática:

Vamos modularizar o código abaixo:

```
def msg_boas_vindas():
    print('Seja Muito Bem Vindo ao Meu Programa')

def msg_para_o_usuario(nome):
    print(f'{nome}, espero que esteja tendo uma boa experiência...')
```

Salve este bloco de código com um nome de fácil identificação, por exemplo, `boasvindas.py`

Representação visual:



Agora vamos importar esse módulo para dentro de outro arquivo de seu código. Em sua IDE, abra ou crie outro arquivo em branco qualquer, em seguida vamos usar o comando import para importar para dentro desse código o nosso módulo previamente salvo **boasvindas.py**.

```
import boasvindas
```

Em seguida, importado o módulo, podemos dar comandos usando seu nome seguido de uma instrução, por exemplo ao digitarmos boasvindas e inserirmos um ponto, a IDE irá nos mostrar que comandos podemos usar de dentro dele, no nosso caso, usaremos por enquanto a primeira opção, que se refere a função **msg_boas_vindas()**.

```
import boasvindas
```

```
boasvindas.msg_boas_vindas()
```

Se mandarmos rodar o código, o retorno será: **Seja Muito Bem Vindo ao Meu Programa**

Seguindo com o uso de nosso módulo, temos uma segunda função criada que pode receber uma **string** como argumento. Então pela lógica usamos o nome do módulo, “ponto”, nome da função e passamos o argumento. Por exemplo a **string ‘Fernando’**.

```
import boasvindas
```

```
boasvindas.msg_boas_vindas()
boasvindas.msg_para_o_usuario('Fernando')
```

O Retorno será: **Seja Muito Bem Vindo ao Meu Programa**

Fernando, espero que esteja tendo uma boa experiência...

Outro exemplo, supondo que estamos criando uma calculadora, onde para obtermos uma performance melhor (no que diz respeito ao uso de memória), separamos cada operação realizada por essa calculadora em um módulo diferente. Dessa forma, no corpo de nossa aplicação principal podemos nos focar ao código que irá interagir com o usuário e chamar módulos e suas funções enquanto as funções em si só são executadas de forma independente e no momento certo.

Ex: Inicialmente criamos um arquivo de nome `soma.py` que ficará responsável por conter o código da função que irá somar dois números.

```
def soma(num1, num2):
    s = int(num1) + int(num2)
    return s
```

Repare no código, dentro desse arquivo temos apenas três linhas que contém tudo o que precisamos, mas em outras situações, a ideia de modularizar blocos de código é que eles podem ser bastante extensos dependendo de suas funcionalidades. Aqui apenas criamos uma função de nome `soma()` que recebe como parâmetros duas variáveis `num1` e `num2`, internamente a variável temporária `s` realiza a operação de somar o primeiro número com o segundo e guardar em si esse valor. Por fim, esse valor é retornado para que possa ser reutilizado fora dessa função.

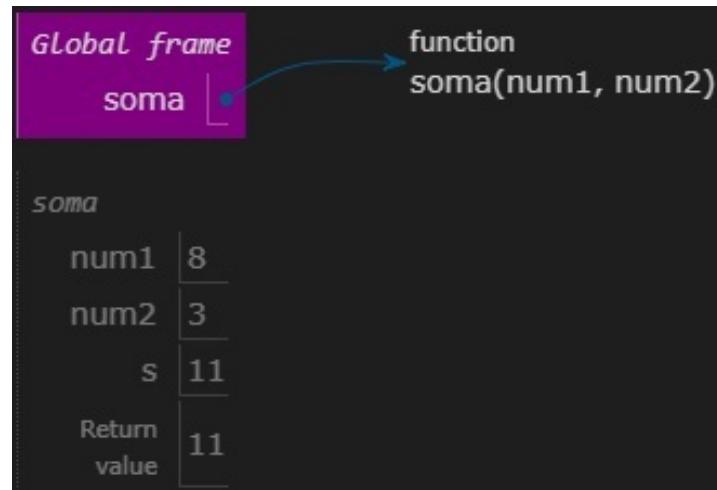
Agora criamos um arquivo de nome `index.py` e dentro dele o seguinte código:

```
import soma

print(f'O resultado da soma é: {soma.soma(8,3)})'
```

Note que inicialmente estamos importando o arquivo `soma.py` (na importação não é necessário o uso da extensão do arquivo). Por fim criamos uma mensagem onde na máscara de substituição estamos executando a função `soma` e passando como parâmetros (para `num1` e `num2`) os números **8** e **3**. Nesse caso o retorno será **11**.

Representação visual:



Caso você queira criar a interação com o usuário, para nesse exemplo anterior, que o mesmo digite os números, esse processo deve ser feito dentro do módulo da função.

Em `soma.py`

```
def soma(num1, num2):
    num1 = input('Digite um número: ')
    num2 = input('Digite outro número: ')
    s = int(num1) + int(num2)
    return s
```

Em **index.py**

```
import soma

print(f'O resultado da soma é: {soma.soma(0,0)})'
```

Apenas note que este código é idêntico ao anterior, e na execução do programa, esses valores **0** e **0** definidos aqui serão substituídos e processados pelos valores que o usuário digitar.

Uma dúvida bastante comum a quem está aprendendo a programar em Python é se precisamos usar a extensão do arquivo quando importamos algum módulo e a resposta é não. Porém, o arquivo de módulo em questão precisa estar na mesma pasta em que estamos trabalhando com os demais arquivos, para que a IDE o ache e consiga o importar. Na verdade, o Python inicialmente procurará o arquivo em questão no diretório atual, se não encontrar ele buscará o arquivo nos diretórios salvos como **PATH**. O recomendável é que você mantenha seus arquivos agrupados em um diretório comum.

Outro conceito importante de ser citado a esta altura é o conceito de Pacotes. Em Python não existe uma grande distinção entre esta forma de se trabalhar com módulos se comparado a outras linguagens de programação. Raciocine que, à medida que você for modularizando seu código, é interessante também dividir categorias de módulos em pastas para melhor organização em geral. Toda vez que criarmos pastas que guardam módulos dentro de si, em Python costumamos chamar isso de um pacote.

Raciocine também que quanto mais modularizado e organizado é seu código, mais fácil será fazer a manutenção do mesmo ou até mesmo a implementação de novas funcionalidades porque dessa forma, cada funcionalidade está codificada em um bloco independente, e assim, algum arquivo corrompido não prejudica muito o funcionamento de todo o programa.

Por fim, na prática, a grande diferença que haverá quando se trabalha com módulos agrupados por pacotes será a forma com que você irá importar os mesmos para dentro de seu código principal.

Importando de módulos:

```
import soma
```

Importando o módulo **soma** (supondo que o mesmo é um arquivo de nome **soma.py** no mesmo diretório).

```
import soma as sm
```

Importando **soma** e o referenciando como **sm** por comodidade.

```
from soma import calculo_soma
```

Importando somente a função **calculo_soma** do módulo **soma**.

Importando de pacotes:

```
import calc.calculadora_soma
```

Importando **calculadora_soma** que faz parte do pacote **calc**. Essa sintaxe **calc . calculadora_soma** indica que **calc** é uma subpasta onde se encontra o módulo **calculadora_soma**.

```
from calc.calculadora_soma import soma
```

Importando a função **soma**, do módulo **calculadora_soma**, do pacote **calc**.

```
from calc.calculadora_soma import soma as sm
```

Importando a função **soma** do módulo **calculadora_soma** do pacote **calc** e a referenciando como **sm**.

Por fim, tenha em mente que todo programa, dependendo claro de sua complexidade, pode possuir incontáveis funcionalidades explícitas ao usuário, mas que o programador definiu manualmente cada uma delas e as categorizou, seguindo uma hierarquia, em funções, módulos, pacotes e bibliotecas. Você pode fazer o uso de toda essa hierarquia criando a mesma manualmente ou usando de funções/módulos/pacotes e bibliotecas já prontas e pré-configuradas disponíveis para o Python. O uso de uma biblioteca inteira ou partes específicas dela tem grande impacto na performance de seu programa, pelas boas práticas de programação, você deve ter na versão final e funcional de seu programa apenas aquilo que interessa, modularizar os mesmos é uma prática comum para otimizar o código.

Ex: Criamos um pacote de nome vendas (uma pasta no diretório com esse nome) onde dentro criamos o arquivo **calc_preco.py** que será um módulo. Na sequência criamos

uma função interna para **calc_preco**. Aqui, apenas para fins de exemplo, uma simples função que pegará um valor e aplicará um aumento percentual.

Em **vendas\calc_preco.py**

```
def aum_preco(preco, porcentagem):
    npreco = preco + (preco * (porcentagem / 100))
    return npreco
```

Em **index.py**

```
import vendas.calc_preco

preco = 19.90
preco_novo = vendas.calc_preco.aum_preco(preco, 4)

print(f'O valor corrigido é: {preco_novo}')
```

Repare que pela sintaxe, inicialmente importamos o módulo **calc_preco** do pacote **vendas**. Em seguida, como já fizemos outras vezes em outros exemplos, usamos essa função passando parâmetros (um valor inicial e um aumento, nesse caso, de 4%) e por fim exibimos em tela o resultado dessa função.

Poderíamos otimizar esse código da seguinte forma:

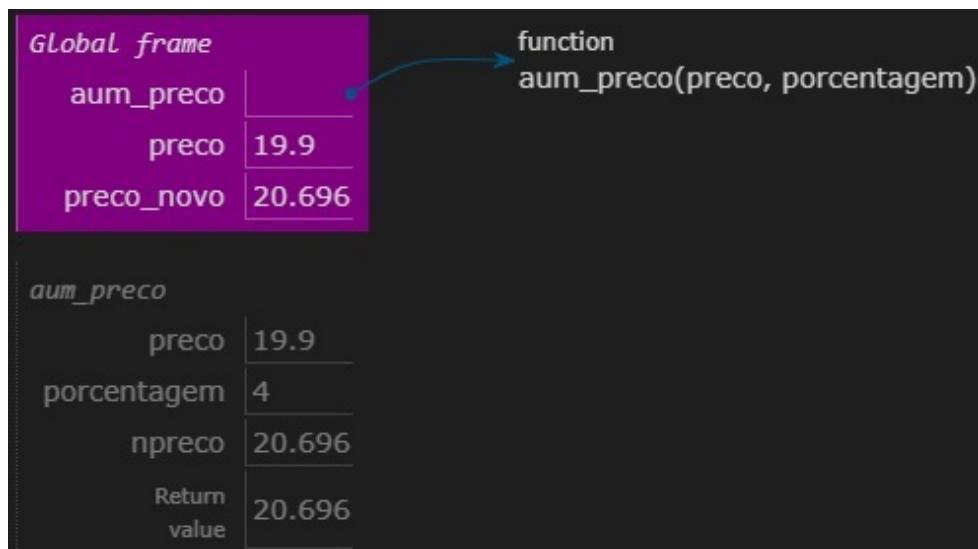
```
from vendas.calc_preco import aum_preco as ap

preco = 19.90
preco_novo = ap(preco, 4)

print(f'O valor corrigido é: {preco_novo}')
```

O resultado seria o mesmo, com um ganho de performance por parte do processamento. Nesse exemplo, o retorno seria: **20.69**.

Representação visual:



Apenas a nível de curiosidade, se você consultar a documentação do Python verá uma infinidade de códigos prontos e pré-configurados para uso. Dependendo o uso, Python tem à disposição bibliotecas inteiras, que contém dentro de si inúmeros pacotes, com inúmeros módulos, com inúmeras funções a disposição, bastando as importar da maneira correta e incorporar em seu código.

21 – Programação orientada a objetos

Classes

Antes de mais nada é importante salientar aqui que, em outras linguagens de programação quando começamos a nos aprofundar nos estudos de classes normalmente há uma separação desde tipo de conteúdo dos demais por estar entrando na área comumente chamada de **orientação a objetos**, em Python não há necessidade de fazer tal distinção uma vez que toda a linguagem em sua forma já é nativa orientada a objetos, logo, progressivamente fomos avançando em seus conceitos e avançaremos muito mais sem a necessidade dessa divisão. Apenas entenda que classes estão diretamente ligadas a programação orientada a objeto, que nada mais é uma abstração de como lidaremos com certos tipos de problemas em nosso código, criando e usando objetos de uma forma mais complexa, a partir de estruturas “moldes”.

Uma classe dentro das linguagens de programação nada mais é do que um objeto que ficará reservado ao sistema tanto para indexação quanto para uso de sua estrutura, é como se criássemos uma espécie de molde de onde podemos criar uma série de objetos a partir desse molde. Assim como podemos inserir diversos outros objetos dentro deles de forma que fiquem instanciáveis (de forma que permita manipular seu conteúdo), modularizados, oferecendo uma complexa, mas muito eficiente maneira de se trabalhar com objetos.

Parece confuso, mas na prática é relativamente simples, tenha em mente que para Python toda variável é um objeto, a forma como lhe instanciamos e/ou irá fazer com que o interpretador o trate com mais ou menos privilégios dentro do código. O mesmo acontece quando trabalharmos com classes. Porém o fato de haver este tipo de dado específico “classe” o define como uma estrutura lógica modelável e reutilizável para nosso código.

Uma classe fará com que uma variável se torne de uma categoria reservada ao sistema para que possamos atribuir dados, valores ou parâmetros de maior complexidade, é como se transformássemos uma simples variável em uma super variável, de maior possibilidade de recursos e de uso. Muito cuidado para não confundir as coisas, o que é comum de acontecer no seu primeiro contato com programação orientada a objetos, raciocine de forma direta que uma classe será uma estrutura molde que poderá comportar dados dentro de si assim como servir de molde para criação de novas variáveis externas.

Pela sintaxe convencionalmente usamos o comando **class** (palavra reservada ao sistema) para especificar que a partir deste ponto estamos trabalhando com este tipo de dado, na sequência definimos um nome para essa classe, onde por convenção, é dado um nome qualquer desde que o mesmo se inicie com letra maiúscula.

Por exemplo:

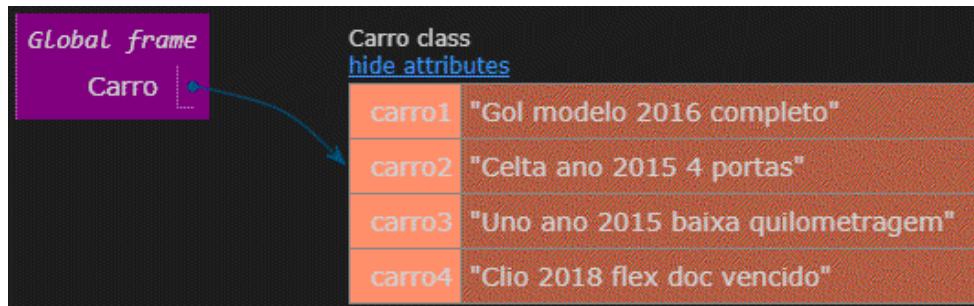
```
class Carro:  
    carro1 = 'Gol modelo 2016 completo'  
    carro2 = 'Celta ano 2015 4 portas'  
    carro3 = 'Uno ano 2015 baixa quilometragem'  
    carro4 = 'Clio 2018 flex doc vencido'  
  
print(Carro.carro1)
```

O Retorno será: **Gol modelo 2016 completo**

Inicialmente foi criada a classe **Carro** (**class** é uma palavra reservada ao sistema e o nome da classe deve ser case sensitive, ou seja, iniciar com letra maiúscula), dentro dessa classe foram inseridos 4 carros com suas respectivas características. Por fim, o comando **print()** mandou exibir em tela a instância **carro1** que pertence a **Carro**. Segundo a lógica do conceito explicado anteriormente, **Carro** é uma super variável que tem **carro1** (uma simples variável) contida nele. O prefixo **class** faz com que o interpretador reconheça essa hierarquia de variáveis e trabalhe da maneira correta com elas.

Abstraindo esse exemplo para simplificar seu entendimento, **Carro** é uma classe/categoria/tipo de objeto, dentro dessa categoria existem diversos tipos de carros, cada um com seu nome e uma descrição relacionada ao seu modelo.

Representação visual:



Para entendermos de forma prática o conceito de uma “super variável”, podemos raciocinar vendo o que ela tem de diferente de uma simples variável contendo dados de qualquer tipo. Raciocine que pelo básico de criação de variáveis, quando precisávamos criar, por exemplo, uma variável **pessoas** que dentro de si, como atributos, guardasse dados como nome, idade, sexo, etc... isto era feito transformando esses dados em uma lista ou dicionário, ou até mesmo criando várias variáveis uma para cada tipo de dado, o que na prática, em programas de maior complexidade, não é nada eficiente.

Criando uma classe **Pessoa**, poderíamos da mesma forma ter nossa variável **pessoa**, mas agora instanciando cada atributo como objeto da classe **Pessoa**, teríamos

acesso direto a esses dados posteriormente, ao invés de buscar dados de índice de um dicionário por exemplo. Dessa forma, instanciar, referenciar, manipular esses dados a partir de uma classe também se torna mais direto e eficiente a partir do momento que você domina essa técnica. Ex:

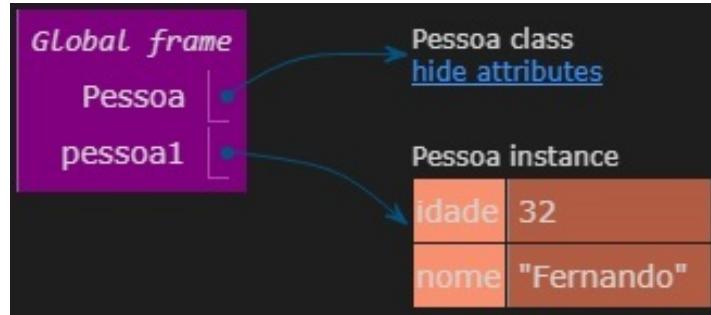
```
class Pessoa:  
    pass  
  
pessoal = Pessoa()  
  
pessoal.nome = 'Fernando'  
pessoal.idade = 32  
  
print(pessoal.nome)  
print(pessoal.idade)
```

Apenas iniciando o entendimento desse exemplo, inicialmente definimos a classe **Pessoa** que por sua vez está vazia de argumentos. Em seguida criamos a variável **pessoal** que recebe como atribuição a classe **Pessoa()**, a partir desse ponto, podemos começar a inserir dados (atributos) que ficarão guardados na estrutura dessa classe. Simplesmente aplicando sobre a variável o comando **pessoal.nome = 'Fernando'** estamos criando dentro dessa variável **pessoal** a variável **nome** que tem como atributo a string '**Fernando**', da mesma forma, dentro da variável **pessoal** é criada a variável **idade = 32**. Note que **pessoal** é uma super variável por conter dentro de si outras variáveis com diferentes dados/valores/atributos... Por fim, passando como parâmetro para nossa função **print()** **pessoal.nome** e **pessoal.idade**.

O retorno será: **Fernando**

32

Representação visual:



No exemplo anterior, bastante básico, a classe em si estava vazia, o que não é o convencional, mas ali era somente para exemplificar a lógica de guardar variáveis e seus respectivos atributos dentro de uma classe. Normalmente a lógica de se usar classes é justamente que elas guardem dados e se necessário executem funções a partir

dos mesmos. Apenas para entender o básico sobre esse processo, analise o bloco de código seguinte:

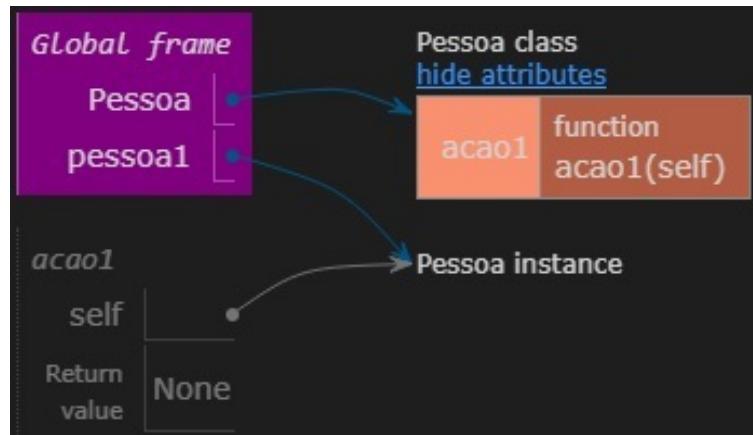
```
class Pessoa:  
    def acao1(self):  
        print('Ação 1 está sendo executada...')  
  
pessoal = Pessoa()  
  
pessoal.acao1()
```

Criada a classe **Pessoa**, dentro dela é definida a função **acao1(self)** que por sua vez exibe em tela uma mensagem. Na sequência é criada a variável **pessoal** que tem como atribuição tudo o que estiver dentro da classe **Pessoa**. Assim como no exemplo anterior adicionamos variáveis com atributos dentro dessa classe a partir da variável inicial, podemos também chamar uma função interna da classe sobre essa variável diretamente aplicando sobre si o nome da função, nesse caso **pessoal.acao1()** executará a função interna na respectiva variável.

O retorno será: **Ação 1 está sendo executada...**

Por fim, note que para tal feito simplesmente referenciamos a variável **pessoal** e chamamos diretamente a função **.acao1()** sobre ela.

Representação visual:



Definindo uma classe

Em Python podemos manualmente definir uma classe respeitando sua sintaxe adequada, seguindo a mesma lógica de sempre, há uma maneira correta de identificar tal tipo de objeto para que o interpretador o reconheça e trabalhe com ele. Basicamente quando temos uma função dentro de uma classe ela é chamada de método dessa classe, que pode executar qualquer coisa. Outro ponto é que quando definimos uma classe manualmente começamos criando um construtor para ela, uma função que ficará reservada ao sistema e será chamada sempre que uma instância dessa classe for criada/usada para que o interpretador crie e use a instância da variável que usa essa classe corretamente. Partindo para prática, vamos criar do zero uma classe chamada **Usuario**:

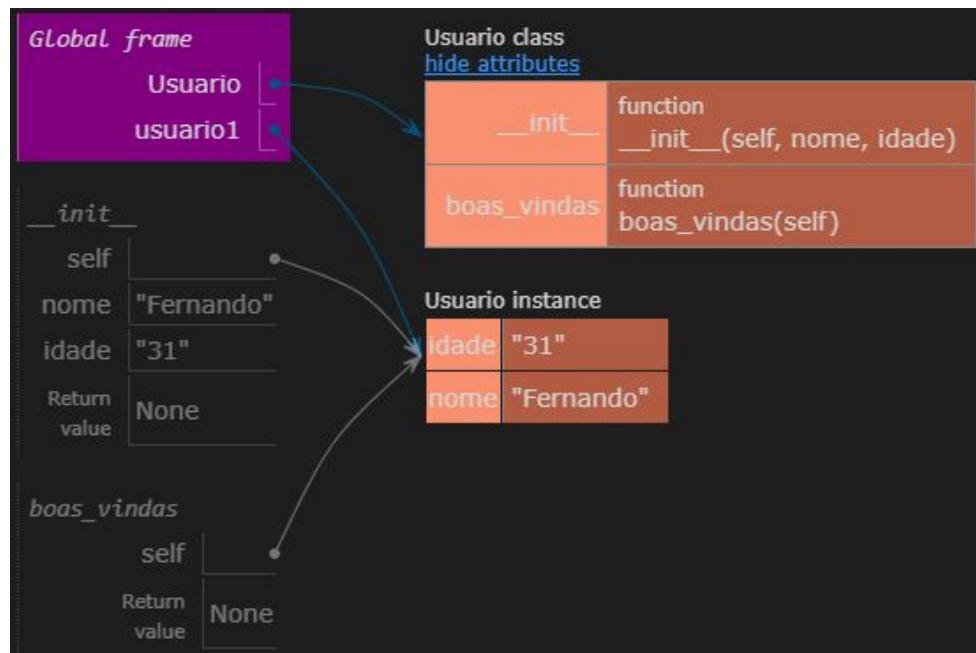
```
class Usuario:  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade  
  
    def boas_vindas(self):  
        print(f'Usuário: {self.nome}, Idade: {self.idade}')  
  
usuario1 = Usuario(nome='Fernando', idade='31')  
usuario1.boas_vindas()  
  
print(usuario1.nome)
```

Repare que o código começa com a palavra reservada **class** seguida do nome da classe que estamos definindo, **Usuario**, que pela sintaxe o nome de uma classe começa com letra maiúscula. Em seguida é declarado e definido o construtor da classe **__init__**, que sempre terá **self** como parâmetro (instância padrão), seguido de quantos parâmetros personalizados o usuário criar, desde que separados por vírgula. Pela indentação existem duas chamadas de **self** para atribuir um nome à variável **nome** e uma idade a variável **idade**. Na sequência há uma nova função apenas chamando a função **print** para uma **string** que interage com as variáveis temporárias declaradas anteriormente. Por fim é criada uma variável **usuario1** que recebe a classe **Usuário** e a atribui dados para **nome** e **idade**. Na execução de **usuario1.boas_vindas()** tudo construído e atribuído até o momento finalmente irá gerar um retorno ao usuário. Há ainda uma linha adicional que apenas chama a função **boas_vindas** para a variável **usuário1**, apenas por convenção mesmo.

O retorno será: **Usuário: Fernando, Idade: 31**

Fernando

Representação visual:



Alterando dados/valores de uma instância

Muito bem, mas e se em algum momento você precisa alterar algum dado ou valor de algo instanciado dentro de uma classe, isto é possível? Sim e de forma bastante fácil, por meio de atribuição direta ou por intermédio de funções que são específicas para manipulação de objetos de classe.

Exemplo de manipulação direta:

```
usuario1 = Usuario(nome='Fernando', idade='31')
usuario1.nome = 'Fernando Feltrin'
usuario1.boas_vindas()

print(usuario1.nome)
```

Aproveitando um trecho do código anterior, repare que na terceira linha é chamada a variável **usuario1** seguida de **.nome** atribuindo uma nova **string** ‘**Fernando Feltrin**’. Ao rodar novamente este código o retorno será: **Usuário: Fernando Feltrin, Idade: 31**

Exemplo de manipulação via função específica:

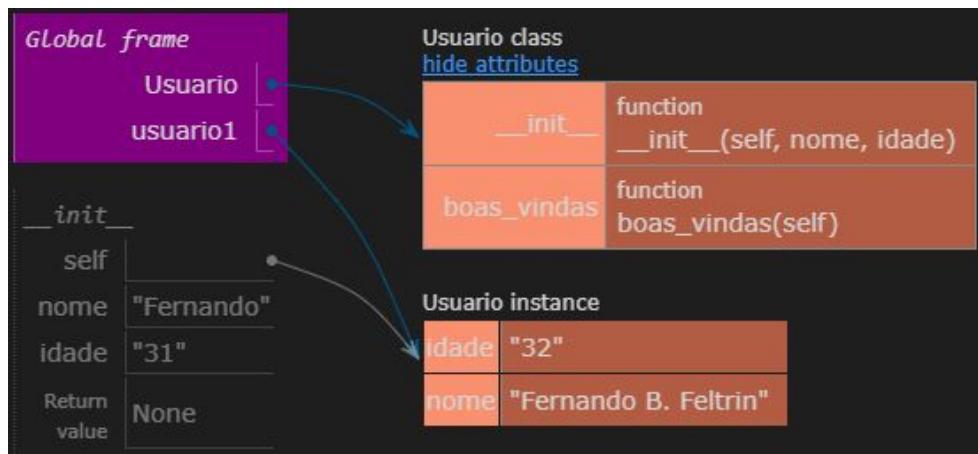
```
usuario1 = Usuario(nome='Fernando', idade='31')
usuario1.nome = 'Fernando Feltrin'

setattr(usuario1, 'nome', 'Fernando B. Feltrin')
delattr(usuario1, 'idade')
setattr(usuario1, 'idade', '32')

print(usuario1.nome)
print(usuario1.idade)
```

Repare que estamos trabalhando ainda no mesmo trecho de código anterior, porém agora na terceira linha temos a função **setattr()** que tem como parâmetro a variável **usuario1** e irá pegar sua instância **nome** e alterar para nova **string** ‘**Fernando B. Feltrin**’. Logo em seguida existe a função **delattr()** que pega a variável **usuario1** e deleta o que houver de dado sob a instância **idade**, logo na sequência uma nova chamada da função **setattr()** irá pegar novamente a variável **usuario1**, agora na instância **idade** e irá atribuir o novo valor, ‘**32**’, em forma de **string** mesmo. Por fim, dando os respectivos comandos **print** para que sejam exibidas as instâncias **nome** e **idade** da variável **usuario1** o retorno serão os valores atualizados. Neste caso o retorno será: **Usuário Fernando B. Feltrin, Idade: 32**

Representação visual:



Em suma, quando estamos trabalhando com classes existirão funções específicas para que se adicionem novos dados (função `setattr(variavel, 'instancia', 'novo dado')`), assim como para excluir um determinado dado interno de uma classe (função `delattr(variavel, 'instancia')`).

Aplicando recursividade

O termo recursividade em linguagens de programação diz respeito à quando um determinado objeto ou função chamar ele mesmo dentro da execução de um bloco de código executando sobre si um incremento.

Imagine que você tem um objeto com dois parâmetros, você aplicará uma condição a ser atingida, porém para deixar seu código mais enxuto você passará funções que retornam e executam no próprio objeto. Por exemplo:

```
def imprimir(maximo, atual):
    if atual >= maximo:
        return
    print(atual)
    imprimir(maximo, atual + 1)

imprimir(10, 1)
```

O resultado será:

```
1
2
3
4
5
6
7
8
9
```

Repare que o que foi feito é que definimos um objeto **imprimir** onde como parâmetros (variáveis) temos **maximo** e **atual**, em seguida colocamos a condição de que se **atual** for maior ou igual a **maximo**, pare a execução do código através de seu retorno. Se essa condição não for atingida, imprima **atual**, em seguida chame novamente o objeto **imprimir**, mudando seu segundo parâmetro para **atual + 1**. Por fim, definimos que **imprimir** recebe como argumentos (o que iremos atribuir a suas variáveis) **10** para **maximo** e **1** para **atual**. Enquanto **atual** não for maior ou igual a **10** ele ficará repetindo a execução desse objeto o incrementando em **1**. Em outras palavras, estamos usando aquele conceito de incremento, visto em **for** anteriormente, mas aplicado a objetos.

Existe a possibilidade de deixar esse código ainda mais enxuto em sua recursividade, da seguinte forma:

```
def imprimir(maximo, atual):
    if atual < maximo:
```

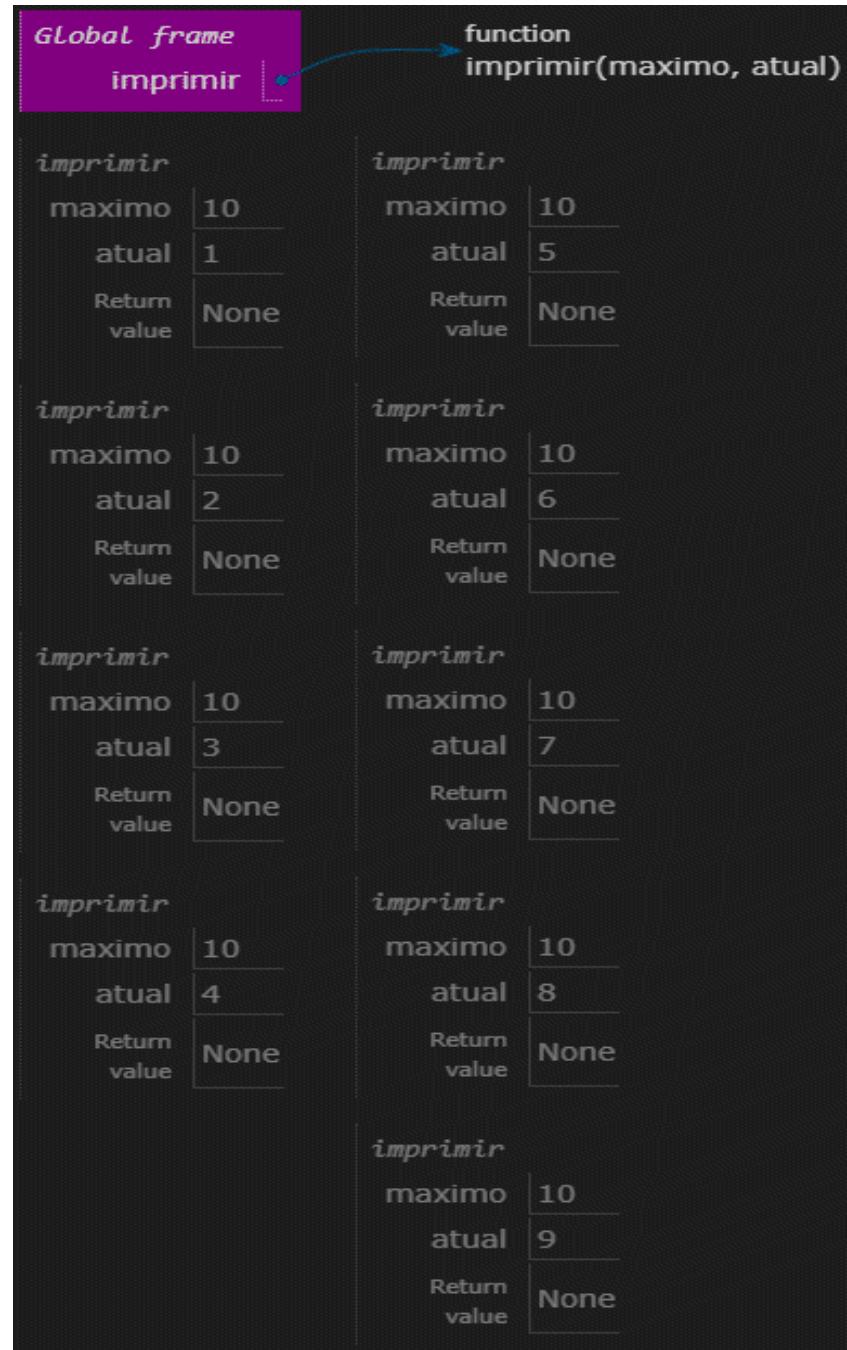
```
print(atual)
imprimir(maximo, atual + 1)
```

```
imprimir(10, 1)
```

O Retorno será: 1

2
3
4
5
6
7
8
9

Representação visual:



Herança

Basicamente quando falamos em herança, a lógica é exatamente a mesma de uma herança “na vida real”, se sou herdeiro de meus pais, automaticamente eu herdo certas características e comportamentos dos mesmos. Em programação a herança funciona da mesma forma, temos um objeto com capacidade de herdar certos parâmetros de outro, na verdade uma classe que herda métodos, parâmetros e outras propriedades de sua classe referência. Imagine que você tem um objeto Carro com uma série de características (modelo, ano, álcool ou gasolina, manual ou automático), a partir dele é criado um segundo objeto Civic, que é um Carro inclusive com algumas ou todas suas características. Dessa forma temos objetos que herdam características de objetos anteriores.

Em Python, como já vimos anteriormente, a estrutura básica de um programa já é pré configurada e pronta quando iniciamos um novo projeto em nossa IDE, fica subentendido que ao começar um projeto do zero estamos trabalhando em cima do método principal do mesmo, ou método `_main_`. Em outras linguagens inclusive é necessário criar tal método e importar bibliotecas para ele manualmente. Por fim quando criamos uma classe que será herdada ou que herdará características, começamos a trabalhar com o método `_init_` que poderá, por exemplo, rodar em cima de `_main_` sem substituir suas características, mas sim adicionar novos atributos específicos.

Na prática, imagine que temos uma classe **Carros**, e como herdeira dela teremos duas novas classes chamadas **Gol** e **Corsa**, que pela sintaxe deve ser montada com a seguinte estrutura lógica:

```
class Carros():
    def __init__(self, nome, cor):
        self.nome = nome
        self.cor = cor

    def descricao(self):
        print(f'O carro: {self.nome} é {self.cor}!')

class Gol(Carros):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)

class Corsa(Carros):
```

```
def __int__(self, nome, cor):
    super().__init__(nome, cor)
```

Repare que primeiro é declarada a classe **Carros**, sem parâmetro nenhum, porém com um bloco de código indentado, ou seja, que pertence a ele. Dentro deste bloco existe o construtor com os tipos de dados aos quais farão parte dessa classe, neste caso, iremos atribuir informações de **nome** e de **cor** a esta classe. Há um segundo método definido onde é uma declaração de descrição, onde será executada a impressão de uma **string** que no corpo de sua sentença irá substituir as máscaras pelas informações de **nome** e **cor** que forem repassadas.

Então é criada a classe **Gol** que tem como parâmetro **Carros**, sendo assim, **Gol** é uma classe filha de **Carros**. Na linha de código indentada existe o construtor e a função **super()** que é responsável por herdar as informações contidas na classe mãe. O mesmo é feito com a classe filha **Corsa**.

A partir disto, de toda essa estrutura montada, é possível criar variáveis que agora sim atribuirão dados de nome e cor para as classes anteriormente criadas.

```
class Carros():
    def __init__(self, nome, cor):
        self.nome = nome
        self.cor = cor

    def descricao(self):
        print(f'O carro: {self.nome} é {self.cor}!')

class Gol(Carros):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)

class Corsa(Carros):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)

gol1 = Gol('Gol 2019 Completo', 'branco')
corsa2 = Corsa('Corsa 2017 2 Portas', 'vermelho')

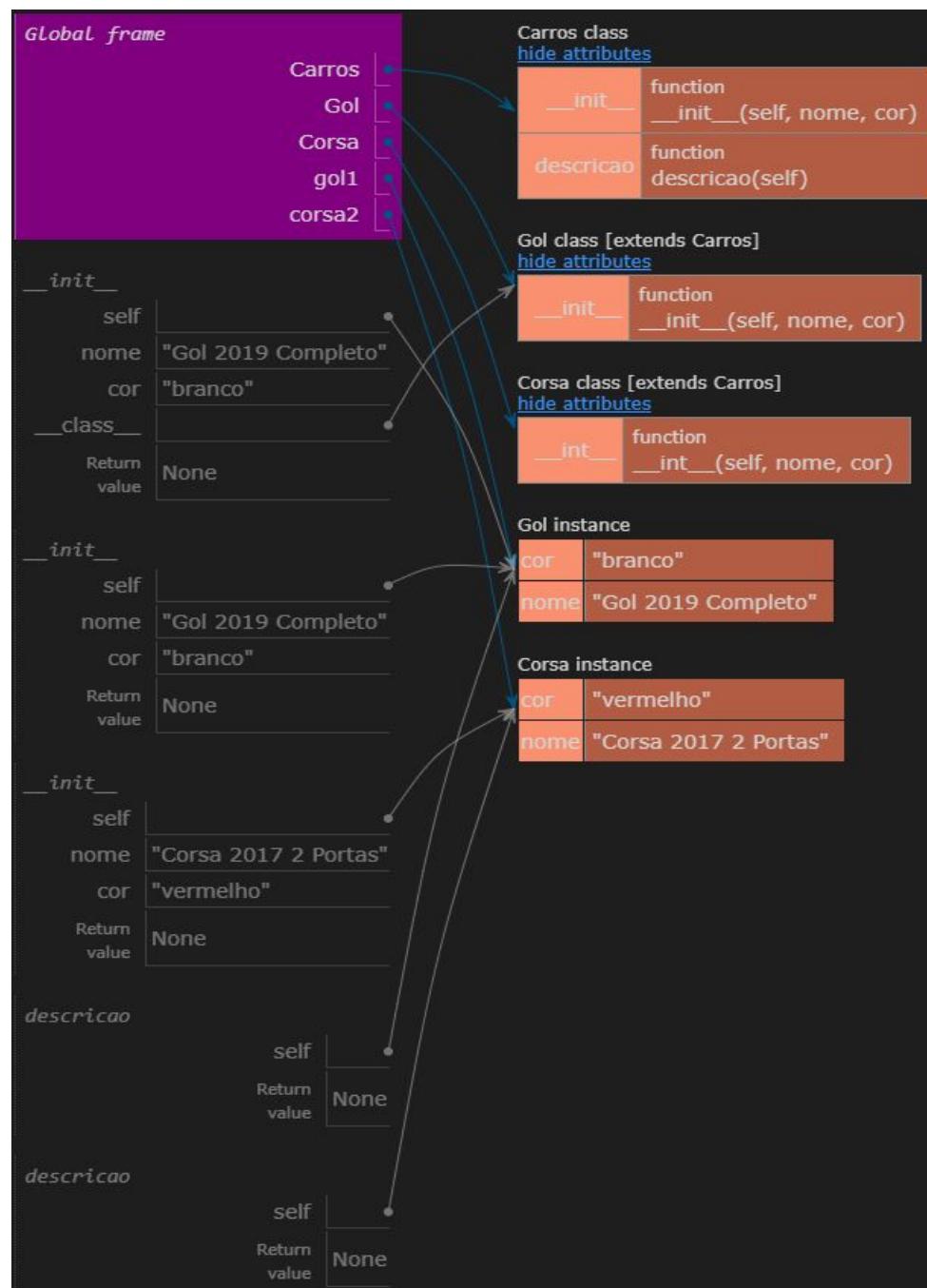
print(gol1.descricao())
print(corsa2.descricao())
```

Repare que é criada a variável **gol1** que recebe a classe **Gol** e passa como parâmetro duas **strings**, que pela sequência serão substituídas por **nome** e **cor** na estrutura da classe. O mesmo é feito criando a variável **corsa2** e atribuindo parâmetros a classe **Corsa**.

Por fim é dado o comando `print`, que recebe como parâmetro a variável `gol1` seguido da função definida `descricao`. `gol1` receberá os dados atribuídos a classe `Gol` e `.descricao` irá executar sua função `print`, substituindo as devidas máscaras em sua sentença.

O retorno será: **O carro: Gol 2019 Completo é branco**

O carro: Corsa 2017 2 Portas é vermelho



Em suma, é possível trabalhar livremente com classes que herdam características de outras, claro que estruturas de tamanha complexidade apenas serão usadas de acordo com a necessidade de seu código possuir de trabalhar com a manipulação de tais tipos de dados.

Polimorfismo

Polimorfismo basicamente é a capacidade de você reconhecer e reaproveitar as funcionalidades de um objeto para uma situação adversa, se um objeto que você já possui já tem as características que você necessita pra quê você iria criar um novo objeto com tais características do zero.

Na verdade, já fizemos polimorfismo no capítulo anterior enquanto usávamos a função **super()** que é dedicada a possibilitar de forma simples que possamos sobrescrever ou estender métodos de uma classe para outra conforme a necessidade.

```
class Carros():
    def __init__(self, nome, cor):
        self.nome = nome
        self.cor = cor

class Gol(Carros):
    def __init__(self, nome, cor):
        super().__init__(nome, cor)
```

Repare que a classe filha **Gol** possui seu método construtor e logo em seguida usa da função **super()** para buscar de **Carros** os parâmetros que ela não tem.

Encapsulamento

O conceito de encapsulamento basicamente é que existe como um objeto e suas funcionalidades ficar encapsulado, fechado de forma que eu consiga instanciá-los sem mesmo conhecer seu conteúdo interior e também é uma forma de tornar um objeto privado, reservado ao sistema de forma que possamos usar livremente porém se o criamos e encapsulamos a ideia é que ele realmente seja imutável.

Principalmente quando usarmos um código de terceiros, muitas das vezes teremos o conhecimento daquele objeto, sua funcionalidade e conseguiremos incorporá-lo em seu código sem a necessidade de alterar ou configurar algo de seu conteúdo, apenas para uso mesmo. Imagine uma cápsula de um determinado remédio, você em grande parte das vezes desconhece a maioria dos componentes químicos que estão ali dentro, porém você sabe o princípio ativo (para que serve) aquele remédio e o toma conforme a necessidade.

Porém, outro fator importante é que em outras linguagens de programação este conceito de tornar-se um objeto privado é muito levado a sério, de forma que em certos casos ele é realmente imutável. Em Python, como ela é uma linguagem dinamicamente tipada, na verdade não existirão objetos com atributos privados ao sistema, até porque não há necessidade disso, você ter o controle sempre é importante, e quando necessário, bastará transformar um **objeto** em um **objeto** para que o mesmo fique reservado ao sistema e ainda seja possível o modificar ou incrementar caso necessário.

```
objeto1 = 'Descrição por extenso'  
#variável/objeto de uso comum.
```

```
__objeto1__ = 'Descrição por extenso'  
#variável/objeto que está reservado ao sistema.
```

```
print(__objeto1__)
```

O retorno será: **Descrição por extenso**

Como em Python tudo é objeto, tudo é dinâmico, e a linguagem coloca o controle total em suas mãos, há a convenção de alguns autores de que o encapsulamento em Python seria mais um aspecto estético (ao bater o olho em qualquer underline duplo __ saber que ali é algo reservado ao sistema) do que de fato ter de se preocupar com o acesso e a manipulação daquele tipo de variável/objeto, dado ou arquivo.

22 – Tracebacks / exceções

Uma das situações que ainda não havíamos comentado, mas que certamente já ocorreu com você ao longo de seus estudos foi o fato de que em certas circunstâncias, quando houver algum erro de lógica ou de sintaxe, o interpretador irá gerar um código de erro. Tais códigos em nossa IDE são chamados de tracebacks e eles tem a finalidade de tentar apontar ao usuário qual é o erro e em que linha do código o mesmo está ocorrendo. Na prática grande parte das vezes um traceback será um erro genérico que apenas irá nos informar o erro, mas não sua solução.

Partindo para camada de software que o usuário tem acesso, nada pior do que ele tentar executar uma determinada função de seu programa e o mesmo apresentar algum erro e até mesmo travar e fechar sozinho. Lembre-se que sempre será um erro humano, de lógica ou de sintaxe. Por exemplo:

```
num1 = 13  
num2 = ad  
  
soma = num1 + num2  
  
print(soma)
```

O retorno será:

```
Traceback (most recent call last):  
  File "C:/Users/Fernando/teste_001.py", line 2, in <module>  
    num2 = ad  
NameError: name 'ad' is not defined
```

Repare no código, declaradas duas variáveis **num1** e **num2** e uma terceira que faz a **soma** das duas anteriores, executado o comando print o retorno será um **traceback**. Analisando o **traceback** ele nos mostra que na execução no nosso atual arquivo (no exemplo **teste_001.py**), na **linha 2** o dado/valor **ad** não é reconhecido como nenhum tipo de dado.

Comandos try, except e finally

Pelas boas práticas de programação, uma solução elegante é prevermos os possíveis erros e/ou na pior das hipóteses apenas mostrar alguma mensagem de erro ao usuário, apontando que está havendo alguma exceção, algo não previsto durante a execução do programa.

Ainda trabalhando em cima do exemplo anterior, o **traceback** se deu pelo fato de estarmos tentando somar um **int** de valor **13** e um dado **ad** que não faz sentido algum para o interpretador.

Através do comando **try** (em tradução livre do inglês = tentar) podemos fazer, por exemplo, com que o interpretador tente executar a soma dos valores daquelas variáveis. Se não for possível, será executado o comando **except** (em tradução livre = exceção), que terá um **print** mostrando ao usuário uma mensagem de erro e caso for possível realizar a operação o mesmo executará um comando **finally** (em tradução livre = finalmente) com o retorno e resultado previsto.

```
try:  
    num1 = int(input('Digite o primeiro numero: '))  
    num2 = int(input('Digite o segundo numero: '))  
  
except:  
    print('Numero invalido, tente novamente;')  
  
finally:  
    soma = int(num1) + int(num2)  
    print(f'O resultado da soma é: {soma}')
```

Repare que o comando inicial deste bloco de código é o **try**, pela indentação, note que é declarada a variável **num1** que pede para o usuário que digite o primeiro número, em seguida é declarada uma segunda variável de nome **num2** e novamente se pede para que o usuário digite o segundo número a ser somado. Como já vimos anteriormente, o comando **input** aceita qualquer coisa que o usuário digitar, de qualquer tamanho, inclusive com espaços e comandos especiais, o **input** encerra sua fase de captação quando o usuário finalmente aperta **ENTER**. Supondo que o usuário digitou nas duas vezes que lhe foi solicitado um número, o código irá executar o bloco **finally**, que por sua vez cria a variável temporária **soma**, faz a devida soma de **num1** e **num2** e por fim exibe em tela uma **string** com o resultado da **soma**. Mas caso ainda no bloco **try** o usuário digitar algo que não é um número, tornando impossível a soma dos mesmos, isto irá gerar uma exceção, o bloco **except** é responsável por capturar esta exceção, ver que algo do bloco anterior está errado, e por fim, neste caso, exibe uma mensagem de

erro previamente declarada. Este é um exemplo de calculadora de soma de dois números onde podemos presumir que os erros que ocorrerão são justamente quando o usuário digitar algo fora da normalidade.

Importante salientar que se você olhar a documentação do Python em sua versão 3 você verá que existem vários muitos tipos de erro que você pode esperar em seu código e por fins de performance (apenas por curiosidade, Python 3 oferece reconhecimento a 30 tipos de erro possíveis), junto do comando **except**: você poderia declarar o tipo de erro para poupar processamento. Supondo que é um programa onde o tipo de arquivo pode gerar uma exceção, o ideal, pelas boas práticas de programação seria declarar um **except TypeError**: assim o interpretador sabe que é previsto aquele tipo de erro em questão (de tipo) e não testa todos os outros possíveis erros. Porém em seus primeiros programas não há problema nenhum em usar um **except**: que de forma básica chama esta função que irá esperar qualquer tipo de erro dentro de seu banco de dados sintático.

Na prática você verá que é bastante comum usarmos **try** e **except** em operações onde o programa interage com o usuário, uma vez que é ele que pode inserir um dado inválido no programa gerando erro.

23 – Considerações finais

Muito bem, como tudo o que tem um começo tem um fim, e chegamos ao final deste pequeno livro sobre Python. Espero que a leitura deste livro tenha sido tão prazerosa para você quanto foi para mim escrevê-lo. E mais importante do que isso, espero que de fato você tenha aprendido a dar os seus primeiros passos dentro dessa linguagem de programação que é incrível.

Seja por hobby ou para fins profissionais, lembre-se que este é apenas o passo inicial de seu aprendizado de Python, há um mundo de possibilidades esperando por você dentro de todas as áreas da programação em que você pode se especializar.

Agradeço a compra deste material e lhe desejo sucesso em suas novas empreitadas.

Sem mais.

Fernando Feltrin

24 – Introdução / Livro 2 - Programação Orientada a Objetos com Python

A programação orientada à objetos é uma forma avançada de se programar, onde deixamos um pouco de lado o conceito de uma variável ser apenas um simples espaço de memória alocado onde está atribuído um dado ou valor. Na chamada programação orientada a objetos damos poderes a nossas variáveis de forma que elas podem ter em si toda uma estrutura de dados como suporte a variáveis dentro de variáveis, atribuições complexas, métodos e funções, etc... de modo a poder criar interações mais eficientes dentro de nossos blocos de código.

Dessa forma, conseguimos criar estruturas lógicas que podem se alocadas, instanciadas, utilizadas, reutilizadas e até mesmo descartadas conforme a necessidade, inclusive definindo toda uma estrutura hierárquica e de permissões sobre as mesmas.

Em outras palavras, há um enorme ganho de performance neste tipo de programação uma vez que nem todo o código precisa ser carregado e executado integralmente, conforme gatilhos vão acionando ações e funções dentro do código, os objetos responsáveis são acionados dinamicamente. Este conceito na verdade existe fora da programação orientada a objetos, quando modularizamos nossos códigos, porém, nos moldes de programação orientada a objetos teremos além de um ganho real de performance, um mundo de possibilidades de interações entre blocos de código, entre módulos e pacotes, além das interações convencionais que estamos acostumados em programação convencional estruturada.

Sendo assim, embora inicialmente pareça mais complexo, conseguiremos criar programas mais robustos, onde dependendo das suas funcionalidades, quando programadas de forma orientada a objetos e não da maneira convencional estruturada, teremos vantagens implícitas como tornar mais fácil a manutenção de um código ou a implementação de novas funções no mesmo, haja visto que nesses moldes podemos realizar tais alterações diretamente no objeto correspondente.

Para muitos estudantes de programação existe uma barreira inicial a ser quebrada quando se entra nesta parte, uma vez que sua sintaxe muda de forma que iremos criar objetos e estruturas de dados de maior complexidade. Assim como tudo na programação, tenha em mente que cada tópico, cada conceito abordado deve ser entendido e posto em prática, caso contrário, você não aprenderá de fato este tipo de programação.

Por fim, só resta lhe fazer uma breve pergunta: - Você está pronto para começar?

Se a resposta for sim, boa leitura e nos vemos nos capítulos seguintes.

25 - Ambiente de Programação

Se tratando do ambiente o qual estaremos desenvolvendo nossos códigos, há uma grande variedade de IDEs que, cada uma com suas respectivas particularidades, não necessariamente são umas melhores que outras, mas atendem melhor o gosto do próprio programador.

Eu particularmente recomendo fortemente o uso da PyCharm, ou do Visual Studio Code ou do Jupyter Notebook. Porém se você já está familiarizado a outra IDE pode fazer o uso dela sem problemas, o importante é você ter um ambiente confiável onde consiga criar e rodar seus códigos em tempo real.

Caso você seja usuário do Windows deve saber que o Python não vem nativamente instalado em seu sistema operacional, sendo assim, antes mesmo de instalar uma IDE é preciso instalar a última versão estável do Python (no momento da escrita deste livro, versão 3.8.0).

Disponível em: <https://www.python.org/downloads/>



Uma vez instalado corretamente a linguagem Python por meio de seu instalador, podemos prosseguir com a instalação da IDE ao qual faremos o uso.

JetBrains PyCharm

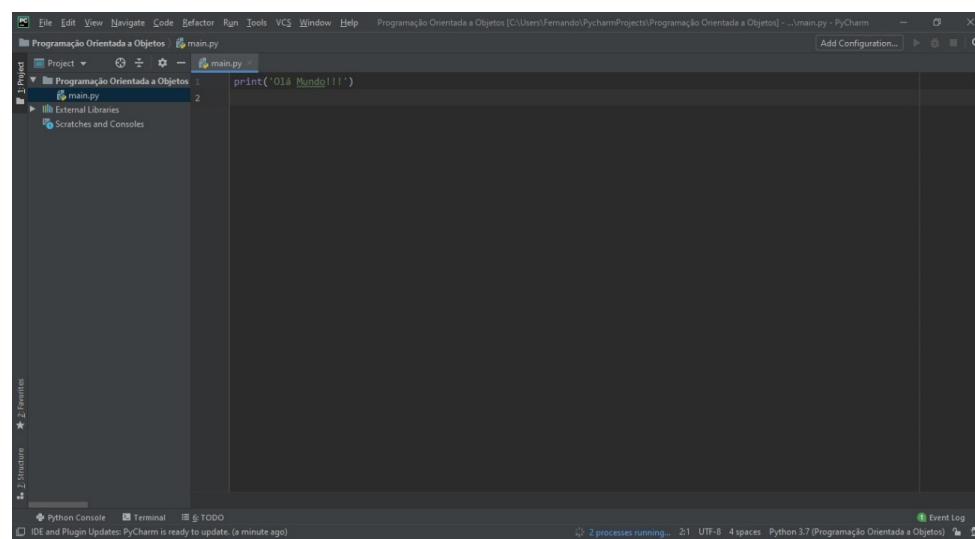
O Pycharm é uma das IDEs mais utilizadas por programadores Python, ela possui uma versão corporativa paga com algumas funcionalidades a mais em relação a sua versão comunitária, que por sua vez é totalmente gratuita. Porém, a versão gratuita da mesma possui todas as funcionalidades que precisará ter à disposição para este tipo de programação, dessa forma, para fins de estudo, é perfeitamente útil a utilização da versão gratuita do PyCharm.

A versão mais recente pode ser baixada em: <https://www.jetbrains.com/pycharm/>

Site



Interface

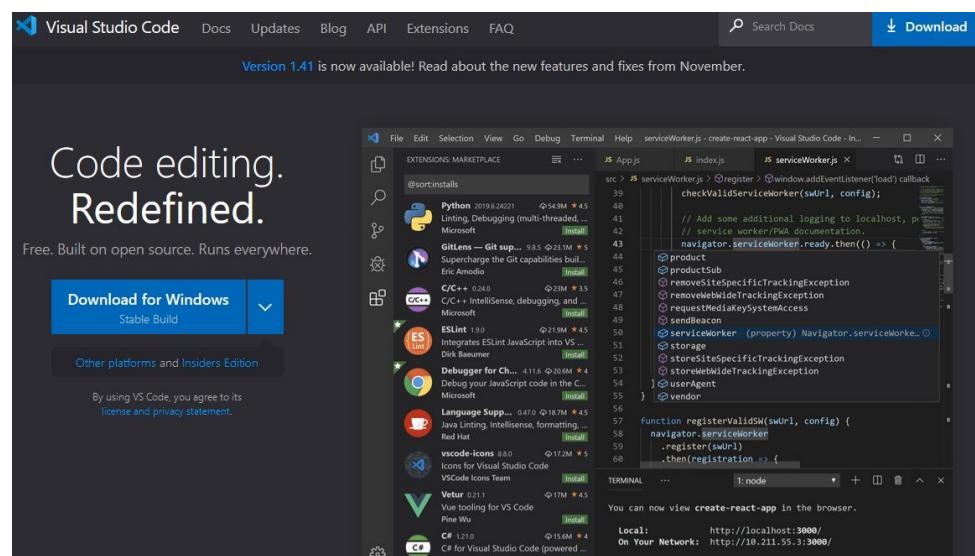


Microsoft Visual Studio Code

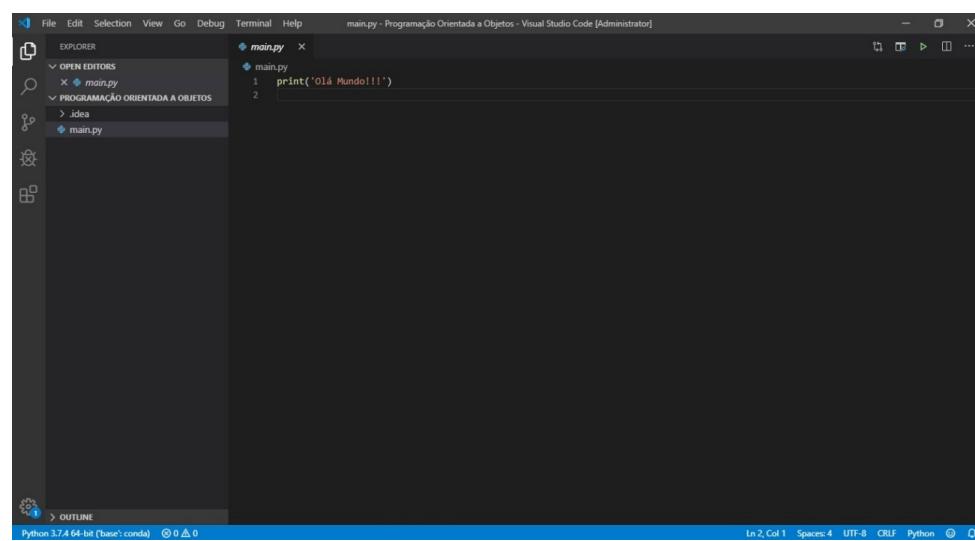
A IDE desenvolvida pela Microsoft não deixa a desejar em nenhum aspecto quando comparada ao PyCharm, inclusive é uma IDE bem aceita pela comunidade e bastante utilizada em função da quantidade de plugins que podem ser instalados na mesma para oferecer outras funcionalidades. É a IDE que estarei utilizando para criar e capturar cada linha de código deste livro, em função do uso de alguns plugins complementares que eu uso normalmente em minhas rotinas.

A versão mais recente pode ser baixada em: <https://code.visualstudio.com/>

Site



Interface



Jupyter Notebook

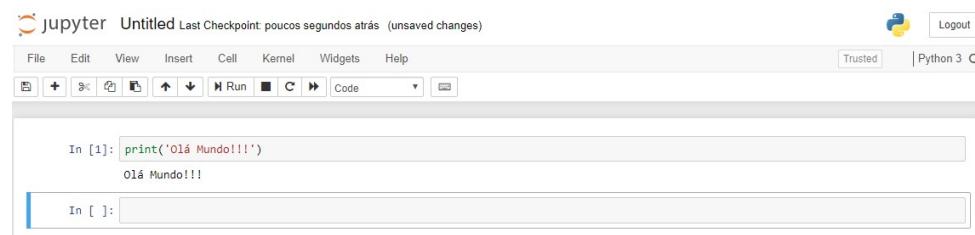
Esta IDE faz parte da suíte Anaconda, muito utilizado por quem está dando os primeiros passos em redes neurais artificiais e machine learning pela característica de suportar a execução individual de blocos de códigos em tempo real, de modo que funções específicas de redes neurais podem ser implementadas e alteradas em tempo real sem afetar o restante do código pré-processado. Este tipo de notebook interativo também é bastante útil por rodar diretamente no navegador e suportar salvar diferentes estados de execução do código.

A versão mais recente pode ser baixada em: <https://jupyter.org/>

Site



Interface



Como mencionado anteriormente, realmente fica a seu critério usar uma dessas IDEs citadas acima ou outra de sua preferência. Os códigos que iremos criar não requerem a instalação de módulos nem bibliotecas adicionais*, estaremos trabalhando em cima do Python 3 puro, em função disso qualquer IDE é capaz de executá-los normalmente.

*em um dos exemplos que serão abordados posteriormente usaremos de uma biblioteca nativa do Python para criação de classes abstratas, não será usada nenhuma biblioteca ou módulo externo, que tenha que ser instalado ou implementado manualmente no Python de nosso sistema.

26 - Programação Orientada a Objetos

Objetos e Classes

Independentemente se você está começando do zero com a linguagem Python ou se você já programa nela ou em outras linguagens, você já deve ter ouvido a clássica frase “em Python tudo é objeto” ou “em Python não há distinção entre variável e objeto”. De fato, uma das características fortes da linguagem Python é que uma variável pode ser simplesmente uma variável assim como ela pode “ganhar poderes” e assumir o papel de um objeto sem nem ao menos ser necessário alterar sua sintaxe.

Por convenção, dependendo que tipo de problema computacional que estaremos abstraindo, poderemos diretamente atribuir a uma variável/objeto o que quisermos, independentemente do tipo de dado, sua complexidade ou seu comportamento dentro do código. O interpretador irá reconhecer normalmente uma variável/objeto conforme seu contexto e será capaz de executá-lo em tempo real.

Outro ponto que precisamos começar a entender, ou ao menos por hora desmistificar, é que na programação orientada a objetos, teremos variáveis/objetos que receberão como atributo uma classe. Raciocine que na programação convencional estruturada você ficava limitado a **variável = atributo**. Supondo que tivéssemos uma variável de nome lista1 e seu atributo fosse uma lista (tipo de dado lista), a mesma estaria explícita como atributo de lista1 e sua funcionalidade estaria ali definida (todas características de uma lista). Agora raciocine que, supondo que houvessemos um objeto de nome mercado1, onde lista1 fosse apenas uma das características/atributos internas desse objeto, essas demais estruturas estariam dentro de uma classe, que pode inclusive ser encapsulada ou modularizada para separar seu código do restante.

Em outras palavras, apenas tentando exemplificar a teoria por trás de uma classe em Python, uma classe é uma estrutura de dados que pode perfeitamente guardar dentro de si uma infinidade de estruturas de dados, como variáveis e seus respectivos dados/valores/atributos, funções e seus respectivos parâmetros, estruturas lógicas e/ou condicionais, etc... tudo atribuído a um objeto facilmente instanciável, assim como toda estrutura dessa classe podendo ser reutilizada como molde para criação de novos objetos, ou podendo interagir com outras classes.

Como dito anteriormente, não se assuste com estes tópicos iniciais e a teoria envolvida, tudo fará mais sentido à medida que formos exemplificando cada conceito destes por meio de blocos de código.

Variáveis vs Objetos

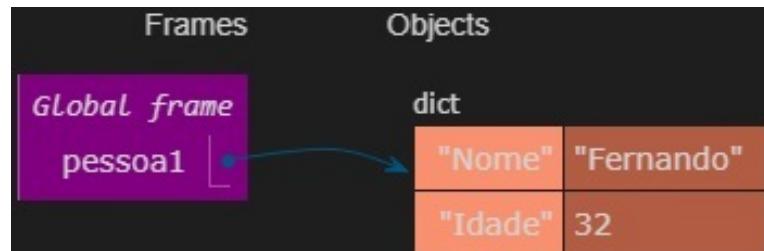
Partindo para a prática, vamos inicialmente entender no código quais seriam as diferenças em um modelo estruturado e em um modelo orientado a objetos. Para esse exemplo, vamos começar do básico apenas abstraindo uma situação onde simplesmente teríamos de criar uma variável de nome **pessoal** que recebe como atributos um nome e uma idade.

Inicialmente raciocine que estamos atribuindo mais de um dado (nome e idade) que inclusive serão de tipos diferentes (para nome uma **string** e para idade um **int**). Convencionalmente isto pode ser feito sem problemas por meio de um dicionário, estrutura de dados em Python que permite o armazenamento de dados em forma de chave / valor, aceitando como dado qualquer tipo de dado alfanumérico desde que respeitada sua sintaxe:

```
pessoal = {'Nome':'Fernando', 'Idade':32}  
print(pessoal)
```

Então, finalmente dando início aos nossos códigos, inicialmente criamos na primeira linha uma variável de nome **pessoal** que por sua vez, respeitando a sintaxe, recebe um dicionário onde as chaves e valores do mesmo são respectivamente **Nome : Fernando** e **Idade : 32**. Em seguida usamos a função **print()** para exibir o conteúdo de **pessoal**.

O retorno será: **Nome : Fernando, Idade : 32**



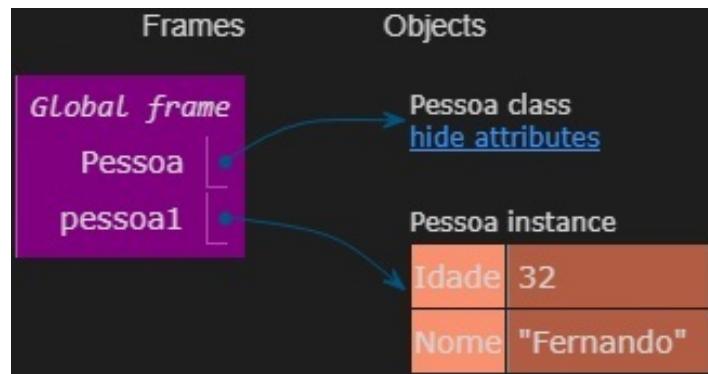
Como dito anteriormente, nesses moldes, temos uma estrutura estática, onde podemos alterar/adicionar/remover dados manipulando o dicionário, mas nada além das funcionalidades padrão de um dicionário.

Partindo para um modelo orientado a objetos, podemos criar uma classe de nome **Pessoa** onde **Nome** e **Idade** podem simplesmente ser variáveis com seus respectivos atributos guardados lá dentro. E não nos limitamos somente a isso, podendo adicionar manualmente mais características a **pessoal** conforme nossa necessidade.

```
class Pessoa:  
    pass  
  
pessoal = Pessoa()  
pessoal.Nome = 'Fernando'  
pessoal.Idade = 32
```

Inicialmente, pela sintaxe Python, para criarmos a estrutura de uma classe de forma que o interpretador faça a sua leitura léxica correta, precisamos escrever **class** seguido do nome da classe, com letra maiúscula apenas para diferenciar do resto. Dentro da classe, aqui nesse

exemplo, colocamos apenas o comando **pass**, dessa forma, esta será inicialmente uma classe “em branco”, por enquanto vazia. Na sequência criamos o objeto de nome **pessoa1** que inicializa a classe **Pessoa** sem parâmetros mesmo, apenas a colocando como atributo próprio. Em seguida, por meio do comando **pessoa1.Nome**, estamos criando a variável **Nome** dentro da classe **Pessoa**, que por sua vez recebe ‘**Fernando**’ como atributo, o mesmo é feito adicionando a variável **Idade** com seu respectivo valor.



Note que em comparação ao código anterior temos mais linhas de código e de maior complexidade em sua lógica sintática, neste exemplo em particular ocorre isto mesmo, porém você verá posteriormente que programar blocos de código dentro de classes assim como modularizar tornará as coisas muito mais eficientes, além de reduzir a quantidade de linhas/blocos de código.

Tratando os dados como instâncias de uma classe, temos liberdade para adicionar o que quisermos dentro da mesma, independentemente do tipo de dado e de sua função no contexto do código.

```
class Pessoa:  

    pass  

pessoa1 = Pessoa()  

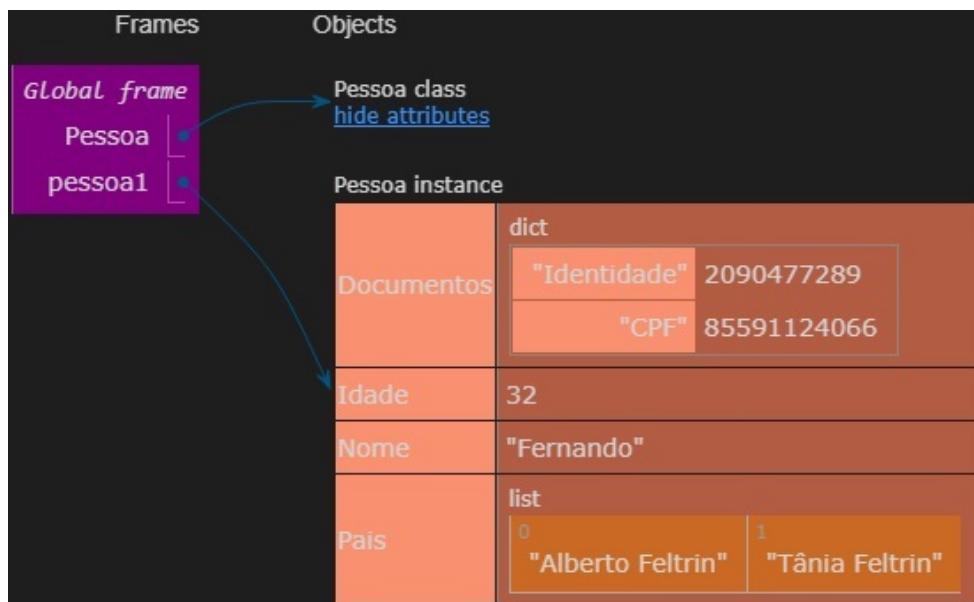
pessoa1.Nome = 'Fernando'  

pessoa1.Idade = 32  

pessoa1.Documentos = {'Identidade':2090477289, 'CPF':85591124066}  

pessoa1.Pais = ['Alberto Feltrin', 'Tânia Feltrin']
```

Repare que seguindo com o exemplo, adicionalmente inserimos um dicionário de nome **Documentos** com valores de **Identidade** e **CPF**, assim como uma lista de nome **Pais** que tem atribuída a si **Alberto Feltrin** e **Tânia Feltrin**.



Toda essa estrutura, guardada na classe **Pessoa**, se fosse ser feita de forma convencional estruturada demandaria a criação de cada variável em meio ao corpo do código geral, dependendo o contexto, poluindo bastante o mesmo e acarretando desempenho inferior em função da leitura léxica do interpretador da IDE.

Sendo assim, aqui inicialmente apenas mostrando de forma visual os primeiros conceitos, podemos começar a entender como se programa de forma orientada a objetos, paradigma este que você notará que será muito útil para determinados tipos de códigos / determinados tipos de problemas computacionais. Nos capítulos subsequentes, gradualmente vamos começar a entender as funcionalidades que podem ser atribuídas a uma classe de um determinado objeto, assim como a forma como instanciaremos dados internos desse objeto para interação com outras partes do código geral.

Outro ponto a levar em consideração neste momento inicial, pode estar parecendo um pouco confuso de entender a real distinção entre se trabalhar com uma variável (programação estruturada) e um objeto (programação orientada a objetos). Para que isso fique bem claro, relembre do básico da programação em Python, uma variável é um espaço de memória alocado onde guardamos algum dado ou valor, de forma estática. Já na programação orientada a objetos, uma variável ganha “super poderes” a partir do momento em que ela possui atribuída a si uma classe, dessa forma ela pode inserir e utilizar qualquer tipo de dado que esteja situado dentro da classe, de forma dinâmica.

Supondo que pela modularização tenhamos um arquivo de nome **pessoa.py**, dentro de si o seguinte bloco de código, referente a uma classe vazia.

```
class Pessoa:  
    pass
```

Agora, em nosso arquivo principal de nome **main.py** realizamos a importação dessa classe **Pessoa** e a atribuímos a diferentes variáveis/objetos.

```
from pessoa import Pessoa  
  
pessoa1 = Pessoa()
```

```
pessoa2 = Pessoa()  
pessoa3 = Pessoa()
```

```
print(pessoa1)  
print(pessoa2)  
print(pessoa3)
```

Diferentemente da programação estruturada, cada variável pode usar a classe **Pessoa** inserindo diferentes dados/valores/atributos a mesma. Neste contexto, a classe **Pessoa** serve como “molde” para criação de outros objetos, sendo assim, neste exemplo, cada variável que inicializa **Pessoa()** está criando toda uma estrutura de dados única para si, usando tudo o que existe dentro dessa classe (neste caso não há nada mesmo, porém poderia ter perfeitamente blocos e mais blocos de códigos...) sem modificar a estrutura de dados da classe **Pessoa**.

```
<pessoa.Pessoa object at 0x000001DAA659AA08>  
<pessoa.Pessoa object at 0x000001DAA6565048>  
<pessoa.Pessoa object at 0x000001DAA84E2208>  
PS C:\Users\Fernando\PycharmProjects\Programação Orientada a Objetos>
```

Por meio da função **print()** podemos notar que cada variável **pessoa** está alocada em um espaço de memória diferente, podendo assim usar apenas uma estrutura de código (a classe **Pessoa**) para atribuir dados/valores a cada variável (ou para mais de uma variável).



Raciocine inicialmente que, a vantagem de se trabalhar com classes, enquanto as mesmas tem função inicial de servir de “molde” para criação de objetos, é a maneira como podemos criar estruturas de código reutilizáveis. Para o exemplo anterior, imagine uma grande estrutura de código onde uma pessoa ou um item teria uma série de características (muitas mesmo), ao invés de criar cada característica manualmente para cada variável, é mais interessante se criar um molde com todas estas características e aplicar para cada nova variável/objeto que necessite destes dados.

Em outras palavras, vamos supor que estamos a criar um pequeno banco de dados onde cada pessoa cadastrada tem como atributos um nome, uma idade, uma nacionalidade, um telefone, um endereço, um nome de pai, um nome de mãe, uma estimativa de renda, etc... Cada dado/valor destes teria de ser criado manualmente por meio de variáveis, e ser recriado para cada nova pessoa cadastrada neste banco de dados... além de tornar o corpo do código principal imenso. É logicamente muito mais eficiente criar uma estrutura de código com todos estes parâmetros que simplesmente possa ser reutilizado a cada novo cadastro, ilimitadamente,

simplesmente atribuindo o mesmo a uma variável/objeto, e isto pode ser feito perfeitamente por meio do uso de classes (estrutura de dados de classe / estrutura de dados orientada a objeto).

Entendidas as diferenças básicas internas entre programação estruturada convencional e programação orientada a objetos, hora de começarmos a de fato criar nossas primeiras classes e suas respectivas usabilidades, uma vez que como dito anteriormente, este tipo de estrutura de dados irá incorporar em si uma série de dados/valores/variáveis/objetos/funções/etc... de forma a serem usados, reutilizados, instanciados e ter interações com diferentes estruturas de blocos de código.

Criando uma classe vazia

Como dito anteriormente, uma classe é uma estrutura de código que permite associar qualquer tipo de atribuição a uma variável/objeto. Também é preciso que se entenda que tudo o que será codificado dentro de uma classe funcionará como um molde para que se criem novos objetos ou a interação entre os mesmos.

Dessa forma, começando a entender o básico das funcionalidades de uma função, primeira coisa a se entender é que uma classe tem uma sintaxe própria que a define, internamente ela pode ser vazia, conter conteúdo próprio previamente criado ou inserir conteúdo por meio da manipulação da variável/objeto ao qual foi atribuída. Além disso, uma classe pode ou não retornar algum dado ou valor de acordo com seu uso em meio ao código.

```
class Pessoa:  
    pass
```

Começando do início, a palavra reservada ao sistema **class** indica ao interpretador que logicamente estamos trabalhando com uma classe. A nomenclatura usual de uma classe deve ser o nome da mesma iniciando com letra maiúscula. Por fim, especificado que se trata de uma classe, atribuído um nome a mesma, é necessário colocar dois pontos para que seja feita sua indentação, muito parecida com uma função comum em Python.

```
class Pessoa:  
    pass  
  
pessoal = Pessoa()  
pessoal.nome = 'Fernando'  
  
print(pessoal.nome)
```

Dando sequência, após criada a classe **Pessoa**, podemos começar a manipular a mesma. Neste exemplo, em seguida é criada no corpo do código geral a variável **pessoal** que inicializa a classe **Pessoa()**, atribuindo a si todo e qualquer conteúdo que estiver inserido nessa classe. É necessário colocar o nome da classe seguido de parênteses pois posteriormente você verá que é possível passar parâmetros neste espaço. Na sequência é dado o comando **pessoal.nome = 'Fernando'**, o que pode ser entendido que, será criada dentro da classe uma variável de nome **nome** que recebe como dado a string '**Fernando**'. Executando a função **print()** e passando como parâmetro **pessoal.nome**, o retorno será **Fernando**.



Atributos de classe

Dando mais um passo no entendimento lógico de uma classe, hora de começarmos a falar sobre os atributos de classe. No exemplo anterior criamos uma classe vazia e criamos de fora da classe uma variável de nome **nome** que recebia '**Fernando**' como atributo. Justamente, um atributo de classe nada mais é do que uma variável criada/declarada dentro de uma classe, onde de acordo com a sintaxe Python, pode receber qualquer dado ou valor.

```
class Pessoa:  
    nome = 'Fernando'  
  
pessoa1 = Pessoa()  
  
print(pessoa1.nome)
```

Dessa vez criamos uma classe de nome **Pessoa**, internamente criamos uma variável de nome **nome** que recebe como atributo '**Fernando**'. Em seguida, fora da classe, criamos uma variável de nome **pessoa1** que recebe como atributo a classe **Pessoa**, a partir desse momento, todo conteúdo de Pessoa estará atribuído a variável **pessoa1**. Dessa forma, por meio da função **print()** passando como parâmetro **pessoa1.nome**, o resultado será: **Fernando**, já que a variável interna a classe Pessoa agora pertence a **pessoa1**.



O que fizemos, iniciando nossos estudos, foi começar a entender algumas possibilidades, como: A partir do momento que uma variável no corpo geral do código recebe como atributo uma classe, ela passa a ser um objeto, uma super variável, uma vez que a partir deste momento ela consegue ter atribuída a si uma enorme gama de possibilidades, desde armazenar variáveis dentro de variáveis até ter inúmeros tipos de dados dentro de si. Também vimos que é perfeitamente possível ao criar uma classe, criar variáveis dentro dela que podem ser instanciadas de fora da classe, por uma variável/objeto qualquer. E ainda neste contexto, é importante que esteja bem entendido que, a classe de nosso exemplo Pessoa, dentro de si tem uma variável nome com um atributo próprio, uma vez que esta classe seja instanciada por outra variável, que seja usada como molde para outra variável, toda sua estrutura interna seria transferida também para o novo objeto.

Manipulando atributos de uma classe

Da mesma forma que na programação convencional estruturada, existe a chamada leitura léxica do interpretador, e ela se dá garantindo que o interpretador fará a leitura dos dados em uma certa ordem e sequência (sempre da esquerda para a direita e linha após linha descendente). Seguindo essa lógica, um bloco de código declarado posteriormente pode alterar ou “atualizar” um dado ou valor em função de que na ordem de interpretação a última informação é a que vale. Aqui não há necessidade de nos estendermos muito, mas apenas raciocine que é perfeitamente normal e possível manipular dados de dentro de uma classe de fora da mesma. Porém é importante salientar que quando estamos trabalhando com uma classe, ocorre a atualização a nível da variável que está instanciando a classe, a estrutura interna da classe continua intacta. Por exemplo:

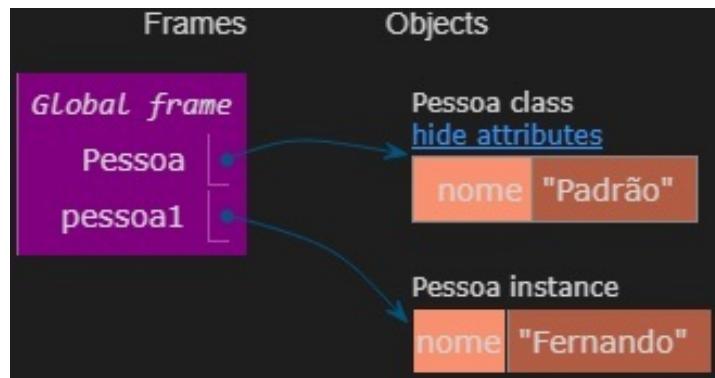
```
class Pessoa:  
    nome = 'Padrão'  
  
pessoal = Pessoa()  
  
print(pessoal.nome)
```

Aproveitando o exemplo anterior, note que está declarada dentro da classe **Pessoa** uma variável de nome **nome** que tem ‘**Padrão**’ como atributo. Logo após criamos uma variável de nome **pessoal** que inicializa a classe e toma posse de seu conteúdo. Neste momento, por meio da função **print()** passando como parâmetro **pessoal.nome** o retorno é: **Padrão**



```
class Pessoa:  
    nome = 'Padrão'  
  
pessoal = Pessoa()  
pessoal.nome = 'Fernando'  
  
print(pessoal.nome)
```

Agora após criar a variável **pessoal** e atribuir a mesma a classe **Pessoa**, no momento que realizamos a alteração **pessoal.nome = ‘Fernando’**, esta alteração se dará somente para a variável **pessoal**. Por meio da função **print()** é possível ver que **pessoal.nome** agora é **Fernando**.



Note que a estrutura da classe, “o molde” da mesma se manteve, para a classe **Pessoa**, nome ainda é ‘Padrão’, para a variável **pessoa1**, que instanciou **Pessoa** e sobrescreveu nome, nome é ‘**Fernando**’. Como dito anteriormente, uma classe pode servir de molde para criação de vários objetos, sem perder sua estrutura a não ser que as alterações sejam feitas dentro dela, e não pelas variáveis/objetos que a instanciam.

Importante salientar que é perfeitamente possível fazer a manipulação de dados dentro da classe operando diretamente sobre a mesma, porém, muito cuidado pois dessa forma, como a classe serve como “molde”, uma alteração diretamente na classe afeta todas suas instâncias. Em outras palavras, diferentes variáveis/objetos podem instanciar uma classe e modificar o que quiser a partir disto, a estrutura da classe permanecerá intacta. Porém quando é feita uma alteração diretamente na classe, essa alteração é aplicada para todas variáveis/objetos que a instancia.

```
class Pessoa:
    nome = 'Padrão'

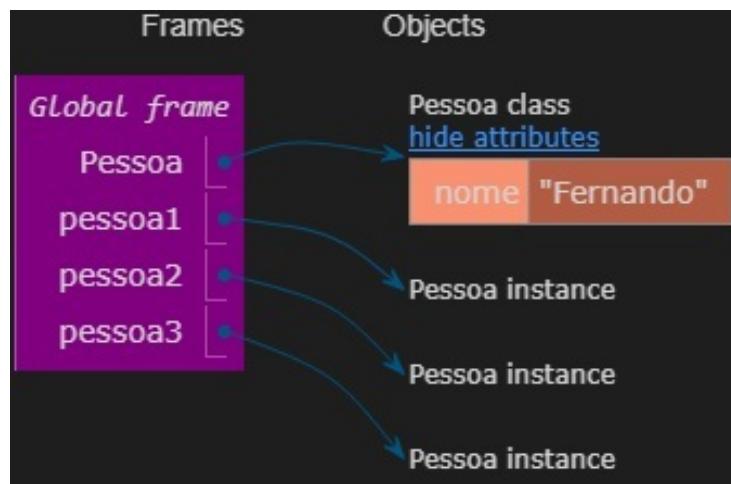
pessoa1 = Pessoa()
pessoa2 = Pessoa()
pessoa3 = Pessoa()

Pessoa.nome = 'Fernando'

print(pessoa1.nome)
print(pessoa2.nome)
print(pessoa3.nome)
```

Aqui seguindo com o mesmo exemplo, porém criando 3 variáveis **pessoa** que recebem como atributo a classe **Pessoa**, a partir do momento que é lida a linha de código **Pessoa.nome = 'Fernando'** pelo interpretador, a alteração da variável **nome** de ‘Padrão’ para ‘Fernando’ é aplicada para todas as instâncias. Sendo assim, neste caso o retorno será:

Fernando
Fernando
Fernando



Em resumo, alterar um atributo de classe via instância altera o valor somente para a instância em questão (somente para a variável/objeto que instanciou), alterar via classe altera diretamente para todas as instâncias que a usar a partir daquele momento.

Métodos de classe

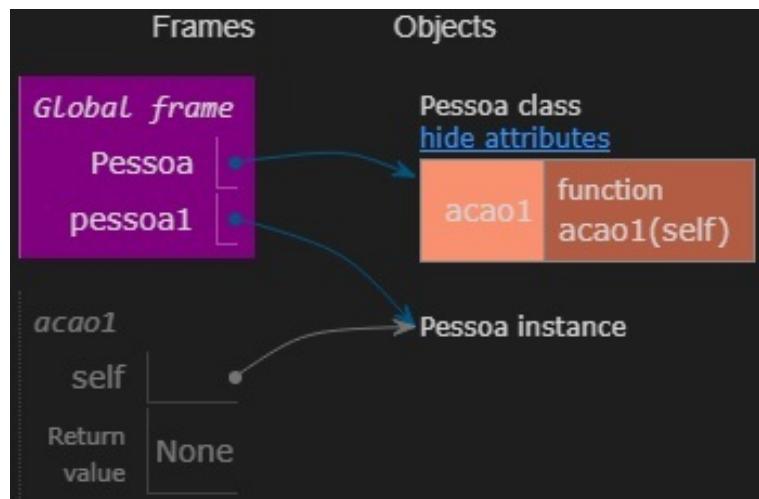
Uma prática bastante comum quando se trabalha com orientação a objetos é criar funções dentro de uma classe, essas funções por sua vez recebem a nomenclatura de métodos de classe. Raciocine que nada disto seria possível na programação estruturada convencional, aqui, como dito anteriormente, você pode criar absolutamente qualquer coisa dentro de uma classe, ilimitadamente. Apenas como exemplo, vamos criar uma simples função (um método de classe) dentro de uma classe, que pode ser executada por meio de uma variável/objeto.

```
class Pessoa:  
    def acao1(self):  
        print('Ação 1 sendo executada...')
```

Todo processo se inicia com a criação da classe **Pessoa**, dentro dela, simplesmente criamos uma função (método de classe) de nome **acao1** que tem como parâmetro **self**. Vamos entender esse contexto, a criação da função, no que diz respeito a sintaxe, é exatamente igual ao que você está acostumado, porém este parâmetro **self** é característico de algo sendo executado dentro da classe. Uma classe implicitamente dá o comando ao interpretador pra que seus blocos de código sejam executados apenas dentro de si, apenas quando instanciados, por uma questão de performance. Posteriormente haverão situações diferentes, mas por hora vamos nos ater a isto. Note que a função **acao1** por sua vez simplesmente exibe uma **string** com uma mensagem pré-programada por meio da função **print()**.

```
class Pessoa:  
    def acao1(self):  
        print('Ação 1 sendo executada...')  
  
pessoal = Pessoa()  
  
print(pessoal.acao1())
```

Da mesma forma do exemplo anterior, é criada a variável **pessoal** que recebe como atributo a classe **Pessoa()**, em seguida, por meio da função **print()** passamos como parâmetro a função **acao1**, sem parâmetros mesmo, por meio de **pessoal.acao1()**. Neste caso o retorno será: **Ação 1 sendo executada...**



Método construtor de uma classe

Quando estamos trabalhando com classes simples, como as vistas anteriormente, não há uma real necessidade de se criar um construtor para as mesmas. Na verdade, internamente esta estrutura é automaticamente gerada, o chamado construtor é criado manualmente quando necessitamos parametrizá-lo manualmente de alguma forma, criando estruturas que podem ser instanciadas e receber atribuições de fora da classe.

Para alguns autores, esse processo é chamado de método construtor, para outros de método inicializador, independente do nome, este é um método (uma função interna) que recebe parâmetros (atributos de classe) que se tornarão variáveis internas desta função, guardando dados/valores a serem passados pelo usuário por meio da variável que instanciar essa classe. Em outras palavras, nesse exemplo, do corpo do código, uma variável/objeto que instanciar a classe **Pessoa** terá de passar os dados referentes a nome, idade, sexo e altura, já que o método construtor dessa classe está esperando que estes dados sejam fornecidos.

```
class Pessoa:  
    def __init__(self, nome, idade, sexo, altura):  
        self.nome = nome  
        self.idade = idade  
        self.sexo = sexo  
        self.altura = altura
```

Ainda trabalhando sobre o exemplo da classe **Pessoa**, note que agora criamos uma função interna de nome **__init__(self)**, palavra reservada ao sistema para uma função que inicializa dentro da classe algum tipo de função. Em sua IDE, você notará que ao digitar **__init__** automaticamente ele criará o parâmetro **(self)**, isto se dá porque tudo o que estiver indentado a essa função será executado internamente e associado a variável que é a instância desta classe (a variável do corpo geral do código que tem esta classe como atributo).

Da mesma forma que uma função convencional, os nomes inseridos como parâmetro serão variáveis temporárias que receberão algum dado ou valor fornecido pelo usuário, normalmente chamados de atributos de classe. Repare que neste exemplo estão sendo criadas variáveis (atributos de classe) para armazenar o **nome**, a **idade**, o **sexo** e a **altura**. Internamente, para que isto se torne variáveis instanciáveis, é criada uma estrutura **self.nomedavariavel** que recebe como atributo o dado ou valor inserido pelo usuário, respeitando inclusive, a ordem em que foram parametrizados.

```
class Pessoa:  
    def __init__(self, nome, idade, sexo, altura):  
        self.nome = nome  
        self.idade = idade  
        self.sexo = sexo  
        self.altura = altura
```

```
pessoal = Pessoa('Fernando', 32, 'M', 1.90)
```

Dessa forma, é criada a variável **pessoal** que recebe como atributo a classe **Pessoa**, que por sua vez tem os parâmetros '**Fernando**', **32**, '**M**', **1.90**. Estes parâmetros serão substituídos

internamente, ordenadamente, pelos campos referentes aos mesmos. O `self` nunca receberá atribuição ou será substituído, ele é apenas a referência de que estes dados serão guardados dentro e para a classe, mas para `nome` será atribuído ‘**Fernando**’, para `idade` será atribuído **32**, para o `sexo` será atribuído **M** e por fim para a `altura` será atribuído o valor de **1.90**.

```
pessoal = Pessoa('Fernando', 32, 'M', 1.90)
```

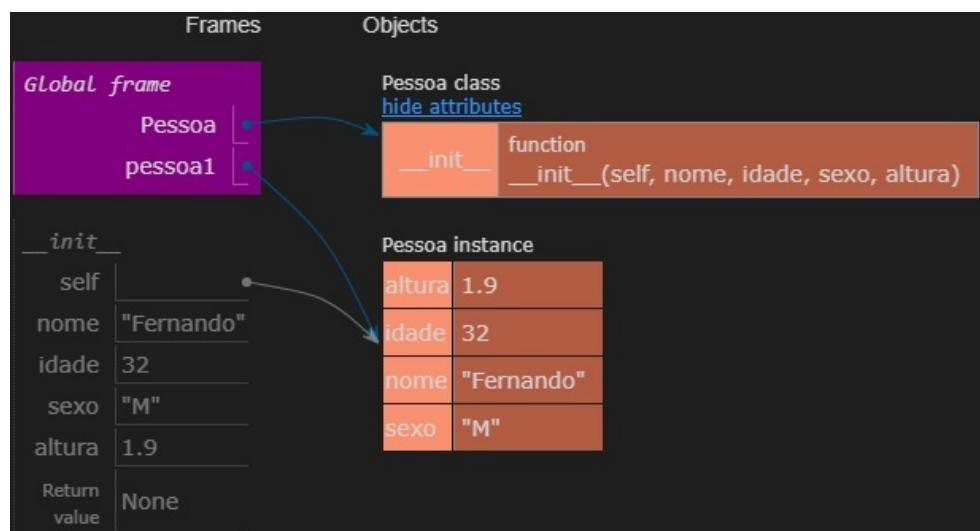
```
print(pessoal.nome, pessoal.idade)
```

A partir deste momento, podemos normalmente instanciarmos qualquer dado ou valor que está guardado na classe **Pessoa**. Por meio da função `print()` passando como parâmetros `pessoal.nome` e `pessoal.idade`, o retorno será: **Fernando, 32**.

```
print(f'Bem vindo {pessoal.nome}, parabéns pelos seus {pessoal.idade} anos!!!')
```

Apenas a nível de curiosidade, a partir do momento que temos variáveis internas a uma classe, com seus devidos atributos, podemos usá-los normalmente de fora da classe, no exemplo acima, criando uma mensagem usando de **f strings** e máscaras de substituição normalmente. Neste caso o retorno será: **Bem vindo Fernando, parabéns pelos seus 32 anos!!!**

A questão da declaração de parâmetros, que internamente farão a composição das variáveis homônimas, devem respeitar a estrutura `self.nomedavariavel`, porém dentro do mesmo escopo, da mesma indentação, é perfeitamente normal criar variáveis independentes para guardar algum dado ou valor dentro de si.



Escopo / Indentação de uma classe

Já que uma classe é uma estrutura que pode guardar qualquer tipo de bloco de código dentro de si, é muito importante respeitarmos a indentação correta desses blocos de código, assim como entender que, exatamente como funciona na programação convencional estruturada, existem escopos os quais tornam itens existentes dentro do mesmo bloco de código que podem ou não estar acessíveis para uso em funções, seja dentro ou fora da classe. Ex:

```
class Pessoa:  
    administrador = 'Admin'  
  
    def __init__(self, nome):  
        self.nome = nome  
  
        msg = 'Classe Pessoa em execução.'  
        print(msg)  
  
    def metodo1(self):  
        print(msg)  
        pass  
  
var1 = Pessoa('Fernando')
```

Neste momento foque apenas no entendimento do escopo, o bloco de código acima tem estruturas que estaremos entendendo a lógica nos capítulos seguintes.

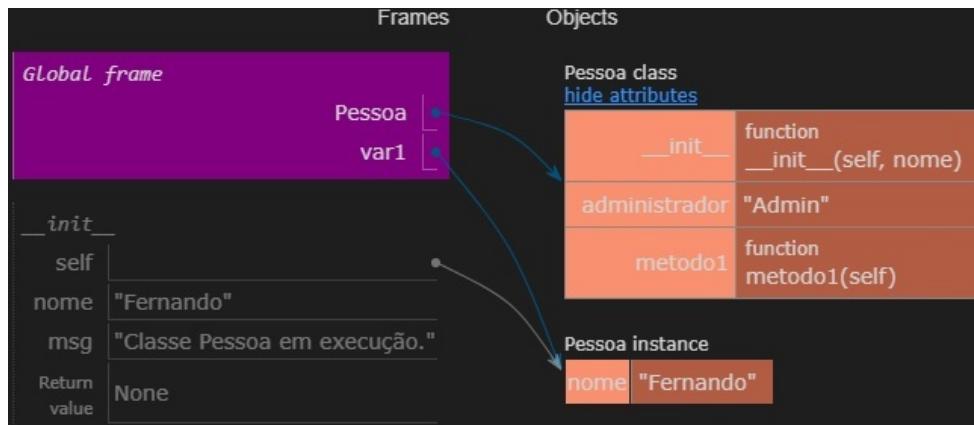
Inicialmente é criada uma classe de nome **Pessoa**, dentro dela existe a variável de nome **administrador** com sua respectiva atribuição '**Admin**', esta variável de acordo com sua localização indentada estará disponível para uso de qualquer função dentro ou fora dessa classe. Em seguida temos o método construtor que simplesmente recebe um nome atribuído pelo usuário e exibe uma mensagem padrão conforme declarado pela variável **msg**. Na sequência é criada uma função chamada **método1** que dentro de si apenas possui o comando para exibir o conteúdo de **msg**, porém, note que **msg** é uma variável dentro do escopo do método construtor, e em função disso ela só pode ser acessada dentro deste bloco de código. Por fim, fora da classe é criada uma variável de nome **var1** que instancia a classe **Pessoa** passando '**Fernando**' como o parâmetro a ser substituído em **self.nome**.

```
print(var1)
```

Por meio da função **print()** passando como parâmetro **var1**, o retorno será a exibição de **msg**, uma vez que é a única linha de código que irá retornar algo ao usuário. Sendo assim o retorno será: **Classe Pessoa em execução**.

```
print(var1.metodo1())
```

Já ao tentarmos executar **msg** dentro da função **metodo1**, o retorno será **NameError: name 'msg' is not defined**, porque de fato, **msg** é uma variável interna de **__init__**, inacessível para outras funções da mesma classe.



Apenas concluindo o raciocínio, a variável **msg**, “solta” dentro do bloco de código construtor, somente é acessível e instanciável neste mesmo bloco de código. Porém, caso quiséssemos tornar a mesma acessível para outras funções dentro da mesma classe ou até mesmo fora dela, bastaria reformular o código e indexá-la ao método de classe por meio da sintaxe **self.msg**, por exemplo.

```
class Pessoa:
    administrador = 'Admin'

    def __init__(self, nome, msg):
        self.nome = nome
        self.msg = msg
        print(msg)

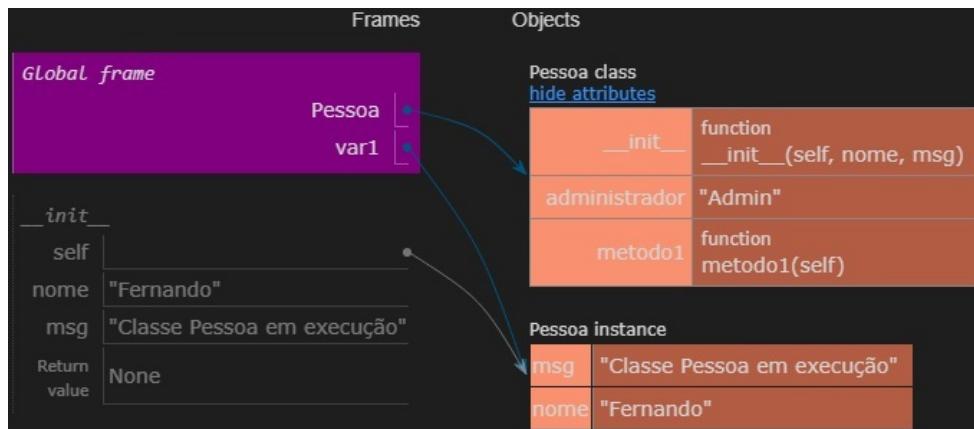
    def metodo1(self):
        print(msg)
        pass
```

```
var1 = Pessoa('Fernando', 'Classe Pessoa em execução')
```

Note que agora **msg** é um dos parâmetros a ser repassados pelo usuário no momento da criação da variável que instancia a classe **Pessoa**.

```
print(var1.metodo1)
```

Dessa forma, por meio da função **print()** fora da classe conseguimos perfeitamente instanciar tal variável uma vez que agora internamente a classe existe a comunicação do método construtor e da função **metodo1**. De imediato isto pode parecer bastante confuso, porém por hora o mais importante é começar a entender as possibilidades que temos quando se trabalha fazendo o uso de classes, posteriormente estaremos praticando mais tais conceitos e dessa forma entendendo definitivamente sua lógica.



Apenas finalizando o entendimento de escopo, caso ainda não esteja bem entendido, raciocine da seguinte forma:

```
class Pessoa:
    pessoa1 = 'Admin'
    #pessoa1 faz parte do escopo global da classe
    #pessoa1 é instanciável por qualquer método

    def __init__(self, pessoa2):
        self.pessoa2 = pessoa2
        #pessoa2 faz parte do escopo do método construtor
        #pessoa2 é acessível dentro e fora deste método
        #pessoa2 pode ser instanciada de fora da classe

    pessoa3 = 'DefaultUser'
    #pessoa3 faz parte do escopo do método construtor
    #pessoa3 é acessível somente dentro deste método
```

Por fim somente fazendo um adendo, é muito importante você evitar o uso de variáveis/objetos de classe de mesmo nome, ou ao menos ter bem claro qual é qual de acordo com seu escopo.

```
class Pessoa:
    nome = 'Padrão'

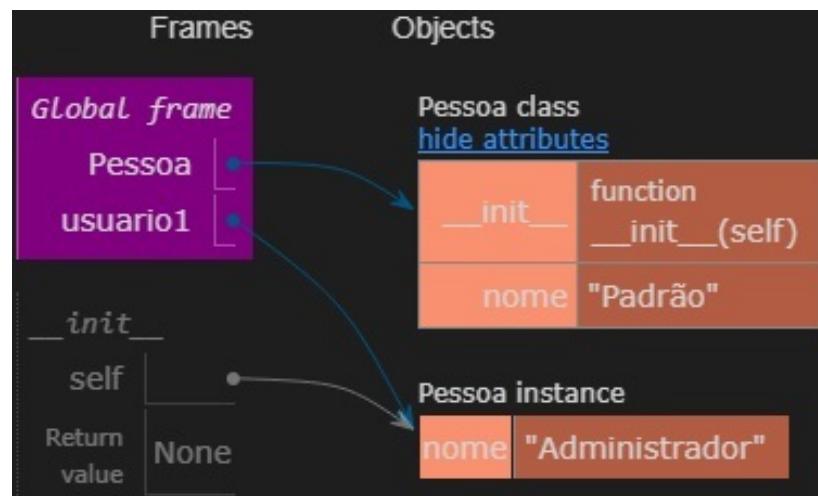
    def __init__(self):
        self.nome = 'Administrador'

usuario1 = Pessoa()

print(usuario1.nome)
print(Pessoa.nome)
```

Apenas simulando uma situação dessas, note que dentro da classe **Pessoa** existe o objeto de classe **nome**, de atributo '**Padrão**', em seguida existe um método construtor que dentro de si também

possui um objeto de classe **nome**, agora de atributo ‘**Administrador**’. Por mais estranho que pareça sempre que há um método construtor dentro de uma classe, esse é lido e interpretado por primeiro, ignorando até mesmo linhas de código anteriores a ele. Pela leitura léxica do interpretador, é feita a leitura linha após linha, porém a prioridade é processar o método construtor dentro da estrutura da classe. Na sequência é criada uma variável que instancia a classe **Pessoa**. Por meio da nossa função **print()**, passando como parâmetro **usuario1.nome** o retorno será **Administrador** (instância do método construtor), passando como parâmetro **Pessoa.nome**, o retorno será **Padrão** (objeto de classe). Então, para evitar confusão é interessante simplesmente evitar criar objetos de classe de mesmo nome, ou ao menos não confundir o uso dos mesmos na hora de incorporá-los ao restante do código.



Classe com parâmetros opcionais

Outra situação bastante comum é a de definirmos parâmetros (atributos de classe) que serão obrigatoriamente substituídos e em contraponto parâmetros que opcionalmente serão substituídos. Como dito anteriormente, uma classe pode servir de “molde” para criar diferentes objetos a partir dela, e não necessariamente todos esses objetos terão as mesmas características. Sendo assim, podemos criar parâmetros que opcionalmente farão parte de um objeto enquanto não farão parte de outro por meio de não atribuição de dados ou valores para esse parâmetro. Basicamente isto é feito simplesmente declarando que os parâmetros que serão opcionais possuem valor inicial definido como **False**. Ex:

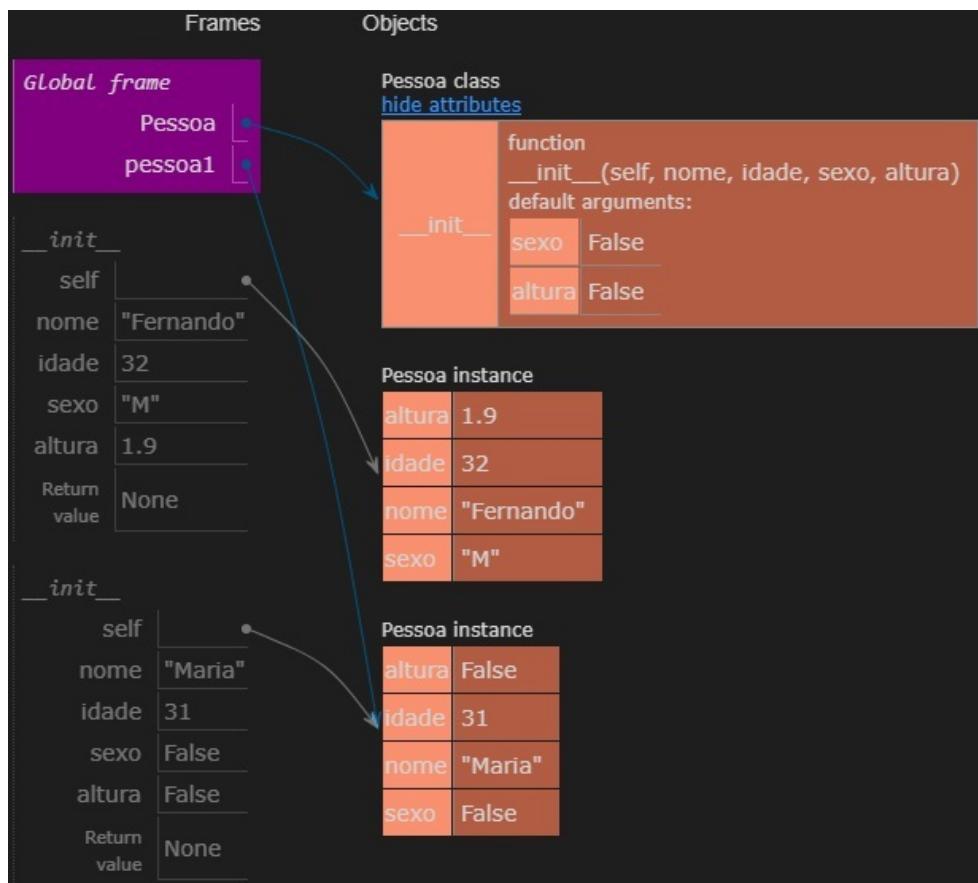
```
class Pessoa:  
    def __init__(self, nome, idade, sexo=False, altura=False):  
        self.nome = nome  
        self.idade = idade  
        self.sexo = sexo  
        self.altura = altura
```

Repare que estamos na mesma estrutura de código do exemplo anterior, porém dessa vez, declaramos que **sexo=False** assim como **altura=False**, o que faz com que esses parâmetros sejam inicializados como **None**. Se o usuário quiser atribuir dados ou valores a estes dados, simplesmente **False** será substituído pelo referente dado/valor, uma vez que **False** é uma palavra reservada ao sistema indicando que neste estado o interpretador ignore essa variável.

```
pessoa1 = Pessoa('Fernando', 32, 'M', 1.90)  
pessoa1 = Pessoa('Maria', 31)
```

```
print(pessoa1.nome, pessoa1.altura)  
print(pessoa2.nome, pessoa2.idade)
```

Dessa forma, continuamos trabalhando normalmente atribuindo ou não dados/valores sem que haja erro de interpretador. Neste caso o retorno será: **Fernando, 1.90. Maria, 31.**



Múltiplos métodos de classe

Uma prática comum é termos dentro de uma classe diversas variáveis assim como mais de uma função (métodos de classe), tudo isso atribuído a um objeto que é “dono” dessa classe. Dessa forma, podemos criar também interações entre variáveis com variáveis e entre variáveis com funções, simplesmente respeitando a indentação dos mesmos dentro da hierarquia estrutural da classe. Por exemplo:

```
class Pessoa:  
    ano_atual = 2019  
  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade
```

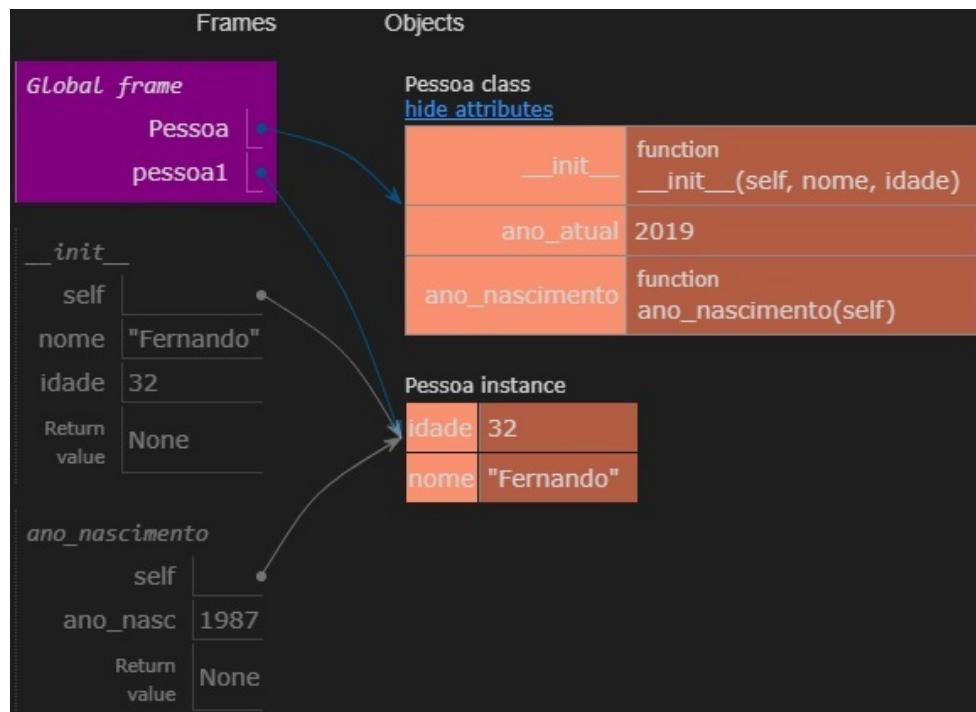
Inicialmente criamos a classe **Pessoa**, dentro dela, criamos uma variável de nome **ano_atual** que recebe como atributo o valor fixo **2019**, **ano_atual** está no escopo geral da classe, sendo acessível a qualquer função interna da mesma. Em seguida, é criado um construtor onde simplesmente é definido que haverão parâmetros para **nome** e **idade** fornecidos pelo usuário, estes, serão atribuídos a variáveis homônimas internas desta função.

```
def __init__(self, nome, idade):  
    self.nome = nome  
    self.idade = idade  
  
def ano_nascimento(self):  
    ano_nasc = self.ano_atual - self.idade  
    print(f'Seu ano de nascimento é {ano_nasc}')
```

Em seguida é criada a função **ano_nascimento**, dentro de si é criada a variável **ano_nasc** que por sua vez faz uma operação instanciando e subtraindo os valores de **ano_atual** e **idade**. Note que esta operação está instanciando a variável **ano_atual** que não faz parte do escopo nem do construtor nem desta função, isto é possível porque **ano_atual** é integrante do escopo geral da classe e está disponível para qualquer operação dentro dela.

```
pessoal = Pessoa('Fernando', 32)  
print(pessoal.ano_nascimento())
```

Na sequência é criada a variável **pessoal** que por sua vez inicializa a classe **Pessoa** passando como parâmetros **'Fernando'**, **32**. Por fim simplesmente pela função **print()** é exibida a mensagem resultante do cruzamento desses dados. Nesse caso o retorno será: **Seu ano de nascimento é 1987**.

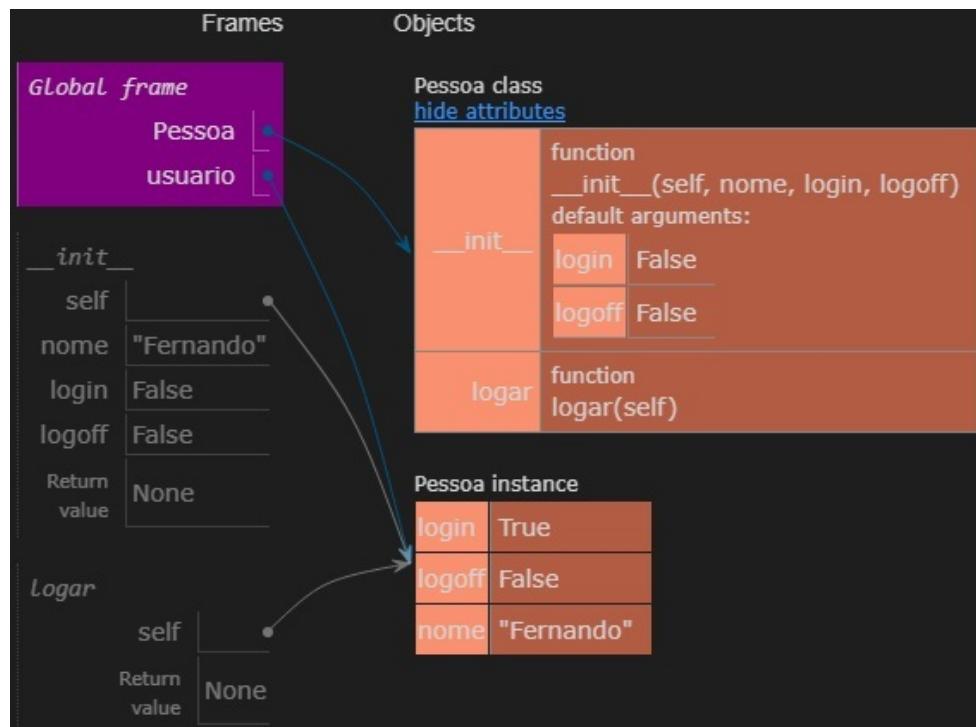


Interação entre métodos de classe

Aprofundando um pouco nossos estudos, podemos criar um exemplo que engloba praticamente tudo o que foi visto até então. Criaremos uma classe, com parâmetros opcionais alteráveis, que internamente realiza a interação entre métodos/funções para retornar algo ao usuário.

```
class Pessoa:  
    def __init__(self, nome, login=False, logoff=False):  
        self.nome = nome  
        self.login = login  
        self.logoff = logoff  
  
    def logar(self):  
        print(f'Bem vindo {self.nome}, você está logado no sistema.')  
        self.login = True  
  
usuario = Pessoa('Fernando')  
usuario.logar()
```

Inicialmente criamos a classe **Pessoa**, em seguida criamos o método construtor da mesma **__init__** que recebe um **nome** atribuído pelo usuário e possui variáveis temporárias reservadas para **login** e **logoff**, por hora desabilitadas. Em seguida criamos as respectivas variáveis internas instanciadas para o construtor por meio da sintaxe **self.nomedavariavel**. Na sequência criamos uma função **logar** que simplesmente exibe em tela uma **f string** com máscara de substituição onde consta o nome atribuído a variável **nome** do método construtor, assim como atualiza o status de **self.login** de **False** para **True**. Fora da classe criamos uma variável chamada **usuario** que instancia a classe **Pessoa** passando como parâmetro '**Fernando**', nome a ser substituído na variável homônima. Por fim, dando o comando de **usuario.logar()**, chamando a função interna **logar** da classe **Pessoa**, o retorno será: **Bem vindo Fernando, você está logado no sistema.**.



Estruturas condicionais em métodos de classe

Apenas um adendo a este tópico, pelo fato de que é bastante comum quando estamos aprendendo este tipo de abstração em programação esquecermos de criar estruturas condicionais ou validações em nosso código. Pegando como exemplo o código acima, supondo que fossem dados vários comandos `print()` sob a mesma função, ocorreria a repetição pura da mesma, o que normalmente não fará muito sentido dependendo da aplicação deste código. Então, lembrando do que foi comentado anteriormente, dentro de uma classe podemos fazer o uso de qualquer estrutura de código, inclusive estruturas condicionais, laços de repetição e validações. Por exemplo:

```
class Pessoa:  
    def __init__(self, nome, login=False, logoff=False):  
        self.nome = nome  
        self.login = login  
        self.logoff = logoff  
  
    def logar(self):  
        print(f'Bem vindo {self.nome}, você está logado no sistema.')  
        self.login = True  
  
usuario = Pessoa('Fernando')  
usuario.logar()  
usuario.logar()
```

O retorno será: **Bem vindo Fernando, você está logado no sistema.**

Bem vindo Fernando, você está logado no sistema.

Porém podemos criar uma simples estrutura que evita este tipo de erro, por meio de uma estrutura condicional que por sua vez para a execução do código quando seu objetivo for alcançado.

```
class Pessoa:  
    def __init__(self, nome, login=False, logoff=False):  
        self.nome = nome  
        self.login = login  
        self.logoff = logoff  
  
    def logar(self):  
        if self.login:  
            print(f'{self.nome} já está logado no sistema')  
            return  
  
        print(f'Bem vindo {self.nome}, você está logado no sistema.')  
        self.login = True  
  
usuario = Pessoa('Fernando')  
usuario.logar()  
usuario.logar()
```

Note que apenas foi criado uma condição dentro do método/função **logar** que, se **self.login** tiver o status como **True**, é exibida a respectiva mensagem e a execução do código para a partir daquele ponto. Então, simulando um erro de lógica por parte do usuário, pedindo para que usuário faça duas vezes a mesma coisa, o retorno será:

Bem vindo Fernando, você está logado no sistema.

Fernando já está logado no sistema

```
class Pessoa:  
    def __init__(self, nome, login=False, logoff=False):  
        self.nome = nome  
        self.login = login  
        self.logoff = logoff  
  
    def logar(self):  
        if self.login:  
            print(f'{self.nome} já está logado no sistema')  
            return  
  
        print(f'Bem vindo {self.nome}, você está logado no sistema.')  
        self.login = True  
  
    def deslogar(self):  
        if not self.login:  
            print(f'{self.nome} não está logado no sistema')  
            return  
        print(f'{self.nome} foi deslogado do sistema')  
        self.login = False
```

Apenas finalizando nossa linha de raciocínio para este tipo de código, anteriormente havíamos criado o atributo de classe **logoff**, porém ainda não havíamos lhe dado um uso em nosso código. Agora simulando que existe uma função específica para deslogar o sistema, novamente o que fizemos foi criar uma função **deslogar** onde consta uma estrutura condicional de duas validações, primeiro ela chega se o usuário não está logado no sistema, e a segunda, caso o usuário esteja logado, o desloga atualizando o status da variável **self.login** e exibe a mensagem correspondente.

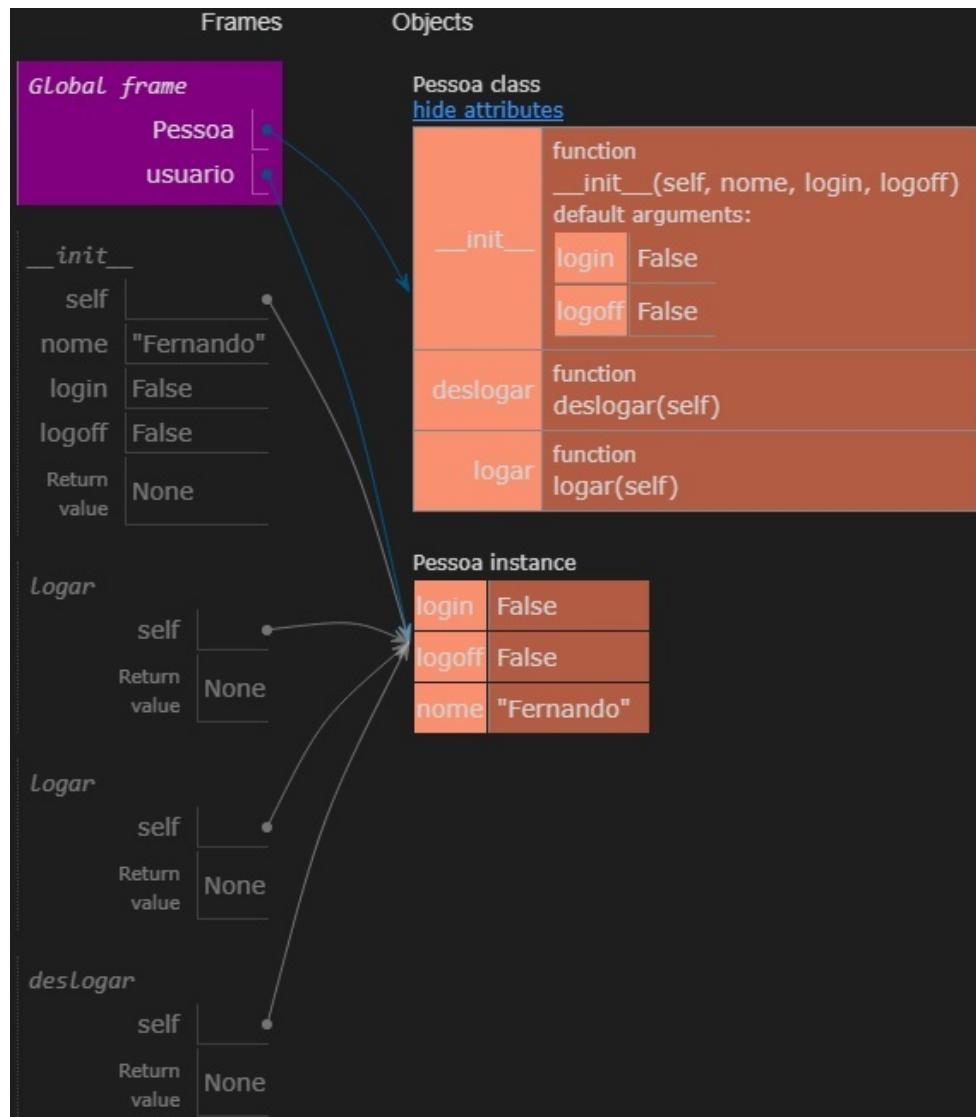
```
usuario = Pessoa('Fernando')  
usuario.logar()  
usuario.logar()  
usuario.deslogar()
```

Novamente simulando o erro, o retorno será:

Bem vindo Fernando, você está logado no sistema.

Fernando já está logado no sistema

Fernando foi deslogado do sistema



Métodos de classe estáticos e dinâmicos

Outra aplicação muito usada, embora grande parte das vezes implícita ao código, é a definição manual de um método de classe quanto a sua visibilidade dentro do escopo da classe. Já vimos parte disso em tópicos anteriores, onde aprendemos que dependendo da indentação do código é criada uma hierarquia, onde determinadas variáveis dentro de funções/métodos de classe podem ser acessíveis e instanciáveis entre funções ou não, dependendo onde a mesma está declarada.

Em certas aplicações, é possível definir esse status de forma manual, e por meio deste tipo de declaração manual é possível fixar o escopo onde o método de classe irá retornar algum dado ou valor. Em resumo, você pode definir manualmente se você estará criando objetos que utilizam de tudo o que está no escopo da classe, ou se o mesmo terá suas próprias atribuições isoladamente.

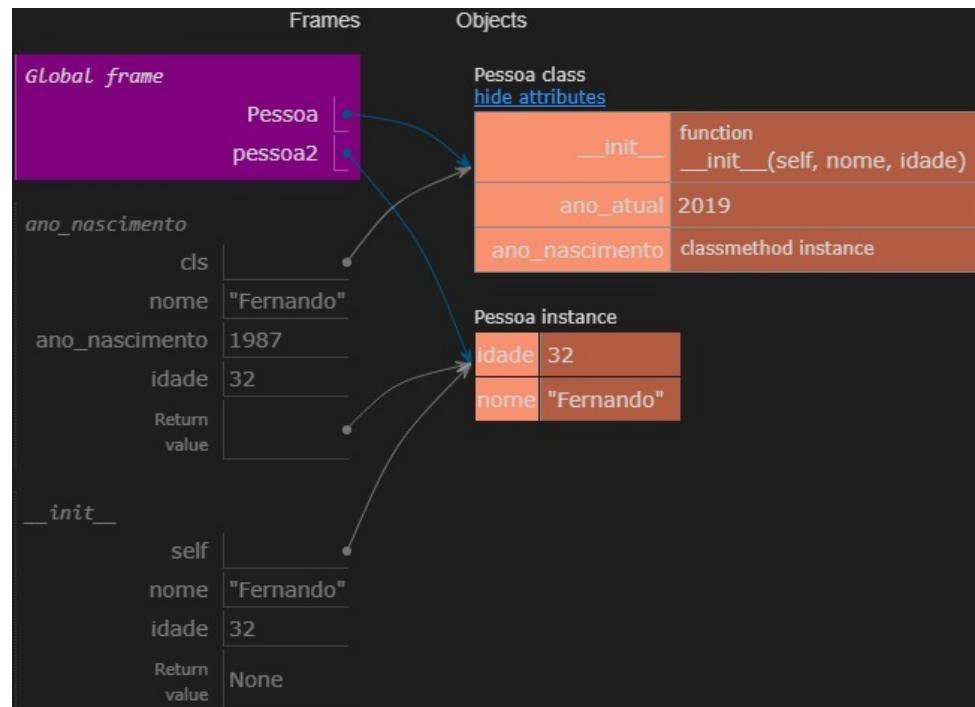
```
class Pessoa:  
    ano_atual = 2019  
  
    def __init__(self, nome, idade):  
        self.nome = nome  
        self.idade = idade  
  
    @classmethod  
    def ano_nascimento(cls, nome, ano_nascimento):  
        idade = cls.ano_atual - ano_nascimento  
        return cls(nome, idade)
```

Da mesma forma que as outras vezes, todo o processo se inicia com a criação da classe, note que estamos reutilizando o exemplo onde calculamos a idade do usuário visto anteriormente. Inicialmente criada a classe **Pessoa**, dentro dela é criada a variável **ano_atual** com **2019** como valor atribuído, logo após, criamos o método construtor da classe que receberá do usuário um **nome** e uma **idade**. Em seguida declaramos manualmente que a classe a seguir é dinâmica, por meio de **@classmethod** (sintaxe igual a de um decorador), o que significa que dentro desse método/função, será realizada uma determinada operação lógica e o retorno da mesma é de escopo global, acessível e utilizável por qualquer bloco de código dentro da classe Pessoa. Sendo assim, simplesmente criamos o método/função **ano_nascimento** que recebe como parâmetro **cls**, **nome**, **ano_nascimento**. Note que pela primeira vez, apenas como exemplo, não estamos criando uma função em **self**, aqui é criada uma variável a critério do programador, independente, que será instanciável no escopo global e internamente será sempre interpretada pelo interpretador como **self**. Desculpe a redundância, mas apenas citei este exemplo para demonstrar que **self**, o primeiro parâmetro de qualquer classe, pode receber qualquer nomenclatura.

```
pessoa2 = Pessoa.ano_nascimento('Fernando', 1987)  
print(pessoa2.idade)
```

Por fim, repare que é criada uma variável de nome **pessoa2** que instancia **Pessoa** e repassa atributos diretamente para o método **ano_nascimento**. Se não houver nenhum erro de sintaxe ocorrerão internamente o processamento das devidas funções assim como a atualização dos dados

em todos métodos de classe. Neste caso, por meio da função `print()` que recebe como parâmetro `pessoa2.idade`, o retorno será: 32



Entendido o conceito lógico de um método dinâmico (também chamado método de classe), hora de entender o que de fato é um método estático. Raciocine que todos métodos de classe / funções criadas até agora, faziam referência ao seu escopo (`self`) ou instância, assim como reservavam espaços para variáveis com atributos fornecidos pelo usuário. Ao contrário disso, se declararmos um método que não possui instâncias como atributos, o mesmo passa a ser um método estático, como uma simples função dentro da classe, acessível e instanciável por qualquer objeto. Um método estático, por sua vez, pode ser declarado manualmente por meio da sintaxe `@staticmethod` uma linha antes do método propriamente dito.

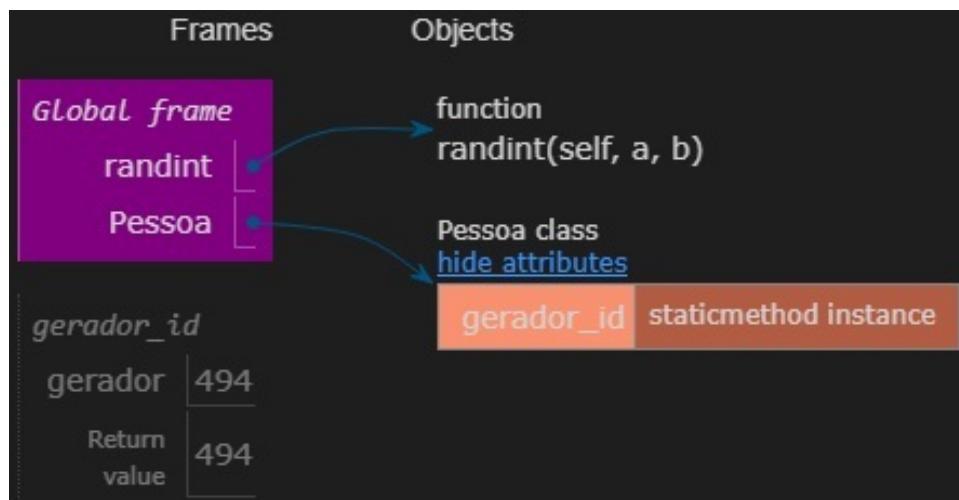
```
from random import randint

class Pessoa:
    @staticmethod
    def gerador_id():
        gerador = randint(100, 999)
        return gerador

print(Pessoa.gerador_id())
```

Mudando um pouco de exemplo, inicialmente importamos da biblioteca `random` o módulo `randint`, muito usado quando é necessário se gerar números inteiros aleatórios. Em seguida é criada a classe `Pessoa`, é declarada que o método `gerador_id()` dentro dela é um método estático pelo `@staticmethod`. O método `gerador_id()` por sua vez simplesmente possui uma variável de nome `gerador` que inicializa o módulo `randint` especificando que seja gerado um número aleatório entre 100 e 999, retornando esse valor para `gerador`. Por fim, por meio da função `print`

) podemos diretamente instanciar como parâmetro a classe **Pessoa** executando seu método estático **gerador_id()**. Neste caso o retorno será um número gerado aleatoriamente entre 100 e 999.



Getters e setters

Anteriormente vimos rapidamente que é perfeitamente possível criarmos estruturas condicionais em nossos métodos de classe, para que dessa forma tenhamos estruturas de código que tentam contornar as possíveis exceções cometidas pelo usuário (normalmente essas exceções são inserções de tipos de dados não esperados pelo interpretador). Aprofundando um pouco mais dentro desse conceito, podemos criar estruturas de validação que contornem erros em atributos de uma classe. Pela nomenclatura usual, um **Getter** obtém um determinado dado/valor e um **Setter** configura um determinado dado/valor que substituirá o primeiro.

```
class Produto:  
    def __init__(self, nome, preco):  
        self.nome = nome  
        self.preco = preco  
  
produto1 = Produto('Processador', 370)  
  
print(produto1.preco)
```

Como sempre, o processo se inicia com a criação de nossa classe, nesse caso, de nome **Produto**, dentro dela criamos um construtor onde basicamente iremos criar objetos fornecendo o **nome** e o **preço** do mesmo para suas respectivas variáveis. De fora da classe, criamos uma variável de nome **produto1** que recebe como atributo a classe **Produto** passando como atributos '**Processador**', **370**. Por meio do comando **print()** podemos exibir por exemplo, o preço de **produto1**, por meio de **produto1.preco**. Neste caso o retorno será **370**

```
class Produto:  
    def __init__(self, nome, preco):  
        self.nome = nome  
        self.preco = preco  
  
    def desconto(self, percentual):  
        self.preco = self.preco - (self.preco*(percentual/100))  
  
produto1 = Produto('Processador', 370)  
produto1.desconto(15)  
  
produto2 = Produto('Placa Mãe', 'R$280')  
produto2.desconto(20)  
  
print(produto1.preco)  
print(produto2.preco)
```

Na sequência adicionamos um método de classe de nome **desconto** responsável por aplicar um desconto com base no percentual definido pelo usuário. Também cadastramos um segundo produto e note que aqui estamos simulando uma exceção no valor atribuído ao preço do mesmo. Já que será executada uma operação matemática para aplicar desconto sobre os valores originais, o interpretador espera que todos dados a serem processados sejam do tipo **int** ou **float**

(numéricos). Ao tentar executar tal função ocorrerá um erro: **TypeError: can't multiply sequence by non-int of type 'float'**.

Em tradução livre: Erro de tipo: Não se pode multiplicar uma sequência de não-inteiros pelo tipo 'float' (número com casas decimais).

Sendo assim, teremos de criar uma estrutura de validação que irá prever esse tipo de exceção e contornar a mesma, como estamos trabalhando com orientação a objetos, mais especificamente tendo que criar um validador dentro de uma classe, isto se dará por meio de **Getters** e **Setters**.

```
#Getter
@property
def preco(self):
    return self.preco_valido

#Setter
@preco.setter
def preco(self, valor):
    if isinstance(valor, str):
        valor = float(valor.replace('R$', ''))
    self.preco_valido = valor
```

Prosseguindo com nosso código, indentado como método de nossa classe **Pessoa**, iremos incorporar ao código a estrutura acima. Inicialmente criamos o que será nosso **Getter**, um decorador (que por sua vez terá prioridade 1 na leitura do interpretador) que cria uma função **preco**, simplesmente obtendo o valor atribuído a **preco** e o replicando em uma nova variável de nome **preco_valido**. Feito isso, hora de criarmos nosso **Setter**, um pouco mais complexo, porém um tipo de validador funcional. Inicialmente criamos um decorador de nome **@preco.setter** (esse decorador deverá ter o nome da variável em questão, assim como a palavra reservada **.setter**). Então criamos a função **preco** que recebe um **valor**, em seguida é criada uma estrutura condicional onde, se o **valor** da instância for do tipo **str**, valor será convertido para **float** assim como os caracteres desnecessários serão removidos. Por fim, **preco_valido** é atualizado com o valor de **valor**.

Código completo:

```
class Produto:
    def __init__(self, nome, preco):
        self.nome = nome
        self.preco = preco

    def desconto(self, percentual):
        self.preco = self.preco - (self.preco*(percentual/100))

    @property
    def preco(self):
        return self.preco_valido

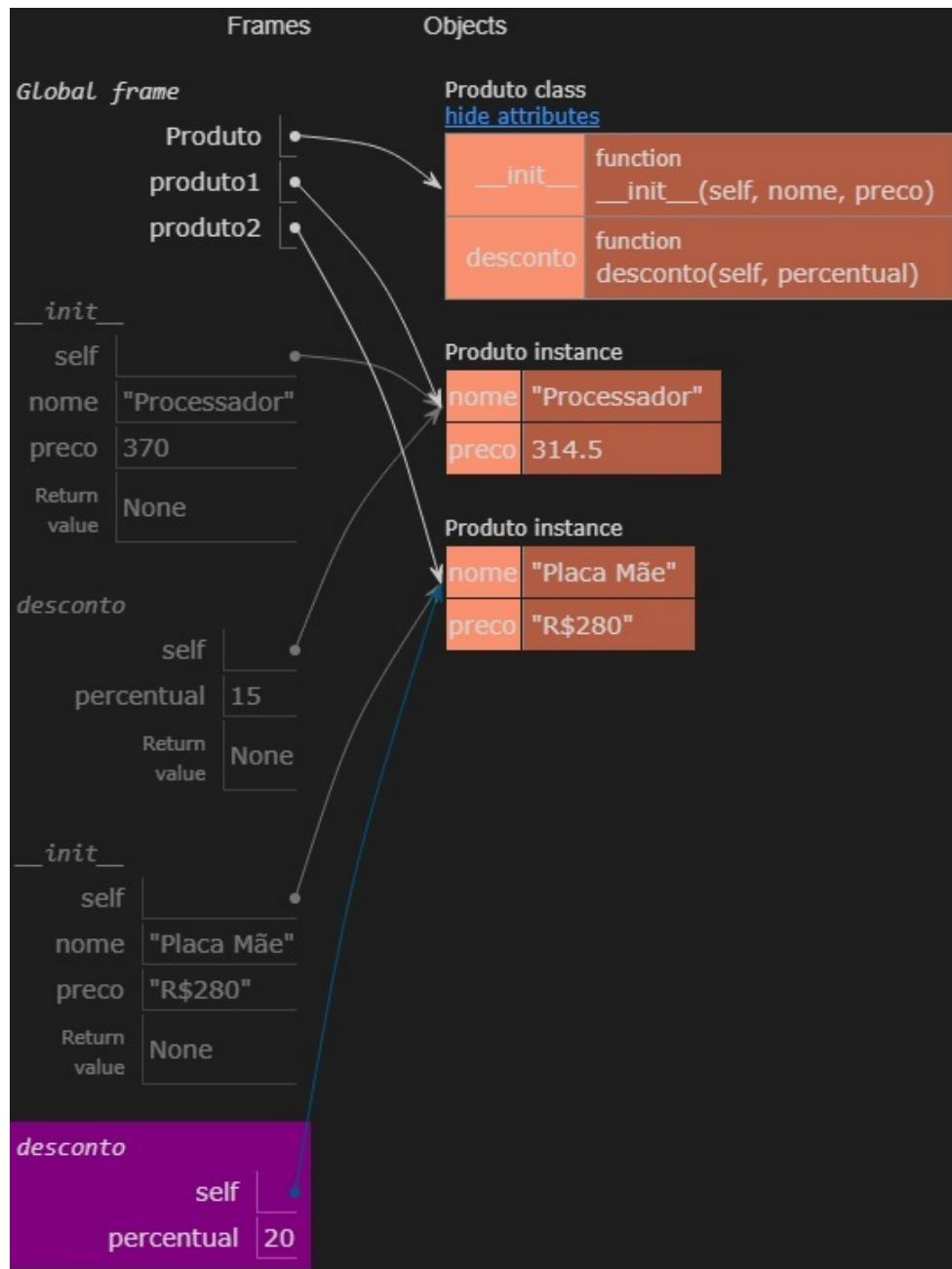
    @preco.setter
    def preco(self, valor):
        if isinstance(valor, str):
```

```
    valor = float(valor.replace('R$', ""))
    self.preco_valido = valor

produto1 = Produto('Processador', 370)
produto1.desconto(15)
produto2 = Produto('Placa M  e', 'R$280')
produto2.desconto(20)

print(produto1.preco)
print(produto2.preco)
```

Realizadas as devidas correções e validações, é aplicado sobre o preço de **produto1** um desconto de 15%, e sobre o preço de **produto2** 20%. Desta forma, o retorno será de: **314.5**
224.0



Encapsulamento

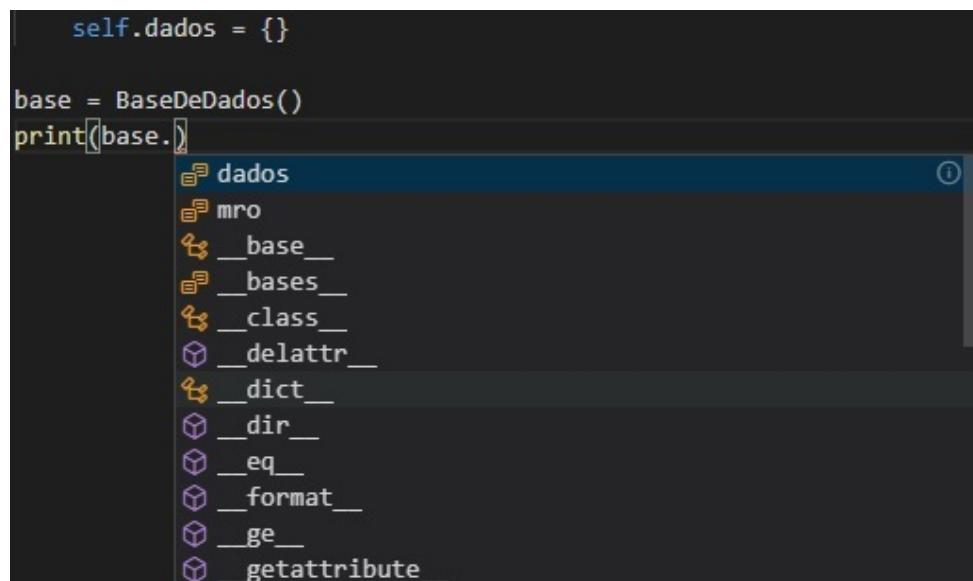
Se você já programa em outras linguagens está habituado a declarar manualmente em todo início de código, se o bloco de código em questão é de acesso público ou privado. Esta definição em Python é implícita, podendo ser definida manualmente caso haja necessidade. Por hora, se tratando de programação orientada a objetos, dependendo da finalidade de cada bloco de código, é interessante você definir as permissões de acesso aos mesmos no que diz respeito a sua leitura e escrita.

Digamos que existam determinadas classes acessíveis e mutáveis de acordo com a finalidade e com sua interação com o usuário, da mesma forma existirão classes com seus respectivos métodos que devem ser protegidos para que o usuário não tenha acesso total, ou ao menos, não tenha como modificá-la. Raciocine que classes e funções importantes podem e devem ser modularizadas, assim como encapsuladas de acordo com sua finalidade, e na prática isto é feito de forma mais simples do que você imagina, vamos ao exemplo:

```
class BaseDeDados:  
    def __init__(self):  
        self.dados = {}  
  
base = BaseDeDados()  
print(base.dados)
```

Repare que aqui, inicialmente, estamos criando uma classe como já estamos habituados, neste caso, uma classe chamada **BaseDeDados**, dentro da mesma há um construtor onde dentro dele simplesmente existe a criação de uma variável na própria instância que pela sintaxe receberá um dicionário, apenas como exemplo mesmo. Fora da classe é criada uma variável de nome **base** que inicializa a classe **BaseDeDados** assim como um comando para exibir o conteúdo de dados.

O ponto chave aqui é, **self.dados** é uma variável, um atributo de classe que quando declarado sob esta nomenclatura, é visível e acessível tanto dentro dessa instância quanto de fora dela.

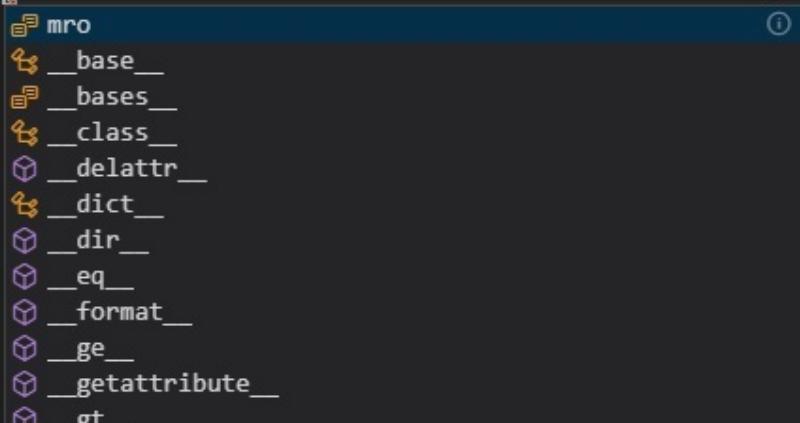


Note que em nossa função `print()` ao passar como parâmetro `base`, sua variável `dados` está disponível para ser instanciada.

A partir do momento que declaramos a variável `dados` pela sintaxe `_dados`, a mesma passa a ser uma variável protegida, não visível, ainda acessível de fora da classe, porém implícita.

```
self._dados = {}

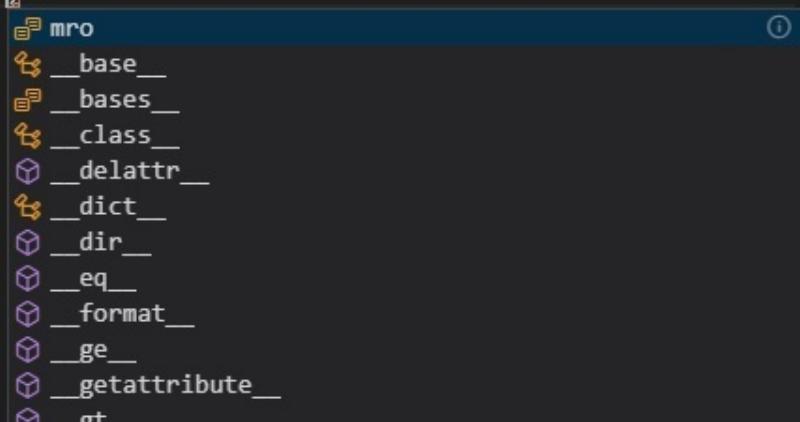
base = BaseDeDados()
print(base.)
```



Repare que ao passar `base` como parâmetro, `dados` não está mais visível, ela pode ser declarada manualmente normalmente, porém supondo que este é um código compartilhado, onde os desenvolvedores tem níveis de acesso diferentes ao código, neste exemplo, `dados` não estaria visível para todos, logo, é um atributo de classe protegido restrito somente a quem sabe sua instância.

```
self._dados = {}

base = BaseDeDados()
print(base.)
```



Por fim, declarando a variável `dados` como `_dados`, a mesma passa a ser privada, e dessa forma, a mesma é inacessível e imutável de fora da classe. Lembre-se que toda palavra em Python, com prefixo `_` (underline duplo) é uma palavra reservada ao sistema, e neste caso não é diferente. Em

resumo, você pode definir a visibilidade de uma variável por meio da forma com que declara, uma underline torna a mesma protegida, underline duplo a torna privada e imutável. Por exemplo:

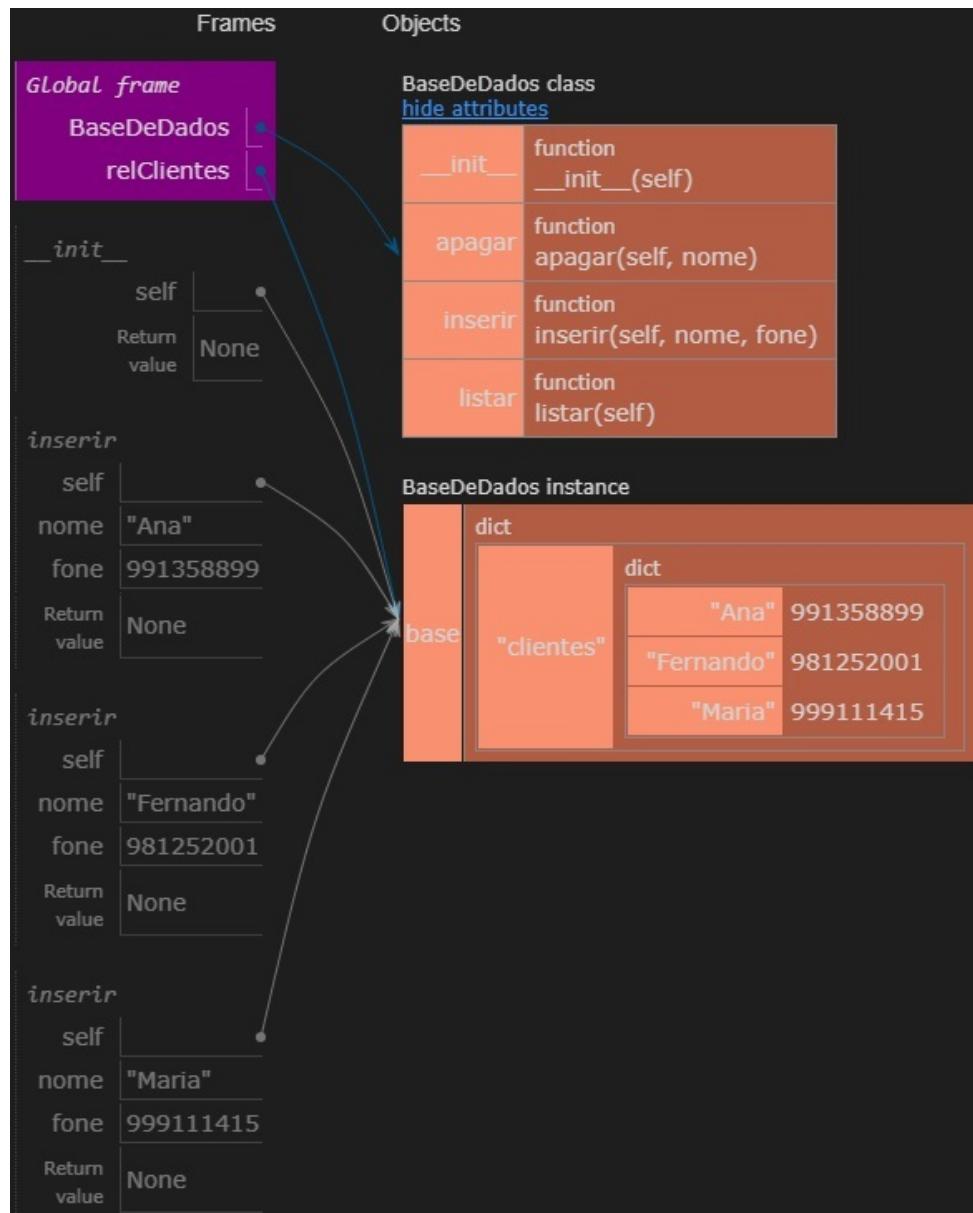
```
class BaseDeDados:  
    def __init__(self):  
        self.base = {}  
  
    def inserir(self, nome, fone):  
        if 'clientes' not in self.base:  
            self.base['clientes'] = {nome:fone}  
        else:  
            self.base['clientes'].update({nome:fone})  
  
    def listar(self):  
        for nome, fone in self.base['clientes'].items():  
            print(nome, fone)  
  
    def apagar(self, nome):  
        del self.base['clientes'][nome]
```

Apenas pondo em prática o conceito explicado anteriormente, supondo que tenhamos um sistema comercial onde sua programação foi feita sem as devidas proteções de código realizadas via encapsulamento. Inicialmente é criada uma classe **BaseDeDados** com um método construtor assim como métodos de classe para **inserir**, **listar** e **apagar** clientes de um dicionário, por meio de seus respectivos **nomes** e **telefones**. Note que nenhum encapsulamento foi realizado (assim como nesse caso, apenas como exemplo, esta base de dados não está modularizada), toda e qualquer parte do código neste momento é acessível e alterável por qualquer desenvolvedor.

```
relClientes = BaseDeDados()  
  
relClientes.inserir('Ana', 991358899)  
relClientes.inserir('Fernando', 981252001)  
relClientes.inserir('Maria', 999111415)  
  
relClientes.listar()
```

Em seguida é criada uma variável que instancia **BaseDeDados()**. Dessa forma, é possível por meio dessa variável usar os métodos internos da classe para suas referidas funções, nesse caso, adicionamos 3 clientes a base de dados e em seguida simplesmente pedimos a listagem dos mesmos por meio do método de classe **listar**. O retorno será:

Ana 991358899
Fernando 981252001
Maria 999111415



Agora simulando uma exceção cometida por um desenvolvedor, supondo que o mesmo pegue como base este código, se em algum momento ele fizer qualquer nova atribuição para a variável **relClientes** o que ocorrerá é que a mesma atualizará toda a classe quebrando assim todo código interno pronto anteriormente.

```
relClientes.base = 'Novo Banco de Dados'  
relClientes.listar()
```

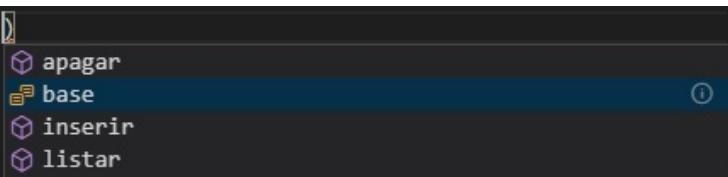
O retorno será:

```
main.py", line 12, in listar for nome, fone in self.base['clientes'].items():  
TypeError: string indices must be integers
```

Isto se deu pelo fato de que, como não havia o devido encapsulamento para proteger a integridade da base de dados, a mesma foi simplesmente substituída pelo novo atributo e a partir desse momento, nenhum bloco de código dos métodos de classe criados anteriormente serão utilizados,

uma vez que todos eles dependem do núcleo de **self.base** que consta no método construtor da classe.

```
print(relClientes.)
```

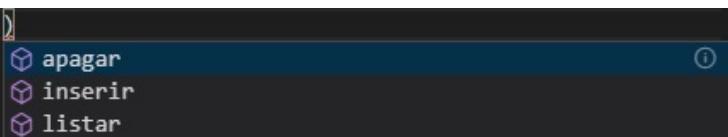


Note que até esse momento ao tentar instanciar qualquer coisa de **relClientes**, entre os métodos aparece inclusive a **base**, que deveria ser protegida.

```
class BaseDeDados:  
    def __init__(self):  
        self.__base = {}
```

Realizando a devida modularização, renomeando todas as referências a **base** para **__base**, a partir deste momento a mesma passa a ser um objeto de classe com as devidas proteções de visualização e contra modificação.

```
print(relClientes.)
```



Executando o mesmo comando **print()** como anteriormente, repare que agora **base** não aparece mais como um objeto instanciável para um usuário desse nível. Ou que ao menos não deveria ser instanciável. Porém, ainda assim é possível aplicar mudanças sobre este objeto, mas como ele está definido de forma a ser imutável, o que acontecerá é que o interpretador irá fazer uma espécie de backup do conteúdo antigo, com núcleo protegido e imutável, que pode ser restaurado a qualquer momento.

```
relClientes.__base = 'Novo Banco de Dados'
```

```
print(relClientes.__base)
```

O interpretador verá que **base** é uma variável protegida, e que manualmente você ainda assim está forçando atribuir a mesma um novo dado/valor. Sendo assim, evitando uma quebra de código, ele irá executar esta função. Neste caso, o retorno será **'Novo Banco de Dados'**.

```
print(relClientes._BaseDeDados__base)
```

Por meio da função **print()**, passando como parâmetro **_BaseDeDados__base** você tem acesso ao núcleo original, salvo pelo interpretador. Dessa forma o resultado será: **{'clientes': {'Ana': 991358899, 'Fernando': 981252001, 'Maria': 999111415}}**

```
BaseDeDados class  
hide attributes
```

__init__	function __init__(self)
apagar	function apagar(self, nome)
inserir	function inserir(self, nome, fone)
listar	function listar(self)

```
BaseDeDados instance
```

__BaseDeDados__base	dict "clientes": dict "Ana": 991358899 "Fernando": 981252001 "Maria": 999111415
__base	"Novo Banco de Dados"

Este é um ponto que gera bastante confusão quando estamos aprendendo sobre encapsulamento de estruturas de uma classe. Visando por convenção facilitar a vida de quem programa, é sempre interessante quando for necessário fazer este tipo de alteração no código deixar as alterações devidamente comentadas, assim como ter o discernimento de que qualquer estrutura de código prefixada com `_` deve ser tratada como um código protegido, e somente como última alternativa realizar este tipo de alterações.

Associação de classes

Associação de classes, ou entre classes, é uma prática bastante comum uma vez que nossos programas dependendo sua complexidade não se restringirão a poucas funções básicas, dependendo do que o programa oferece ao usuário uma grande variedade de funções internas são executadas ou ao menos estão à disposição do usuário para que sejam executadas. Internamente estas funções podem se comunicar e até mesmo trabalhar em conjunto, assim como independentes elas não dependem uma da outra.

Raciocine que esta prática é muito utilizada pois quando se está compondo o código de um determinado programa é natural criar uma função para o mesmo e, depois de testada e pronta, esta pode ser modularizada, ou ser separada em um pacote, etc... de forma que aquela estrutura de código, embora parte do programa, independente para que se faça a manutenção da mesma assim como novas implementações de recursos.

É normal, dependendo a complexidade do programa, que cada “parte” de código seja na estrutura dos arquivos individualizada de forma que o arquivo que guarda a mesma quando corrompido, apenas em último caso faça com que o programa pare de funcionar. Se você já explorou as pastas dos arquivos que compõe qualquer programa deve ter se deparado com milhares de arquivos dependendo o programa. A manutenção dos mesmos se dá diretamente sobre os arquivos que apresentam algum problema.

```
class Usuario:  
    def __init__(self, nome):  
        self.__nome = nome  
        self.__logar = None  
    @property  
    def nome(self):  
        return self.__nome  
    @property  
    def logar(self):  
        return self.__logar  
    @logar.setter  
    def logar(self, logar):  
        self.__logar = logar
```

Partindo para a prática, note que em primeiro lugar criamos nossa classe **Usuario**, a mesma possui um método construtor que recebe um **nome** (com variável homônima declarada como privada), em seguida precisamos realizar a criação de um **getter** e **setter** para que possamos instanciar os referentes métodos de fora dessa classe. Basicamente, pondo em prática tudo o que já vimos até então, esta seria a forma adequada de permitir acesso aos métodos de classe de **Usuario**, quando as mesmas forem privadas ou protegidas.

```
class Identificador:  
    def __init__(self, numero):  
        self.__numero = numero  
    @property  
    def numero(self):
```

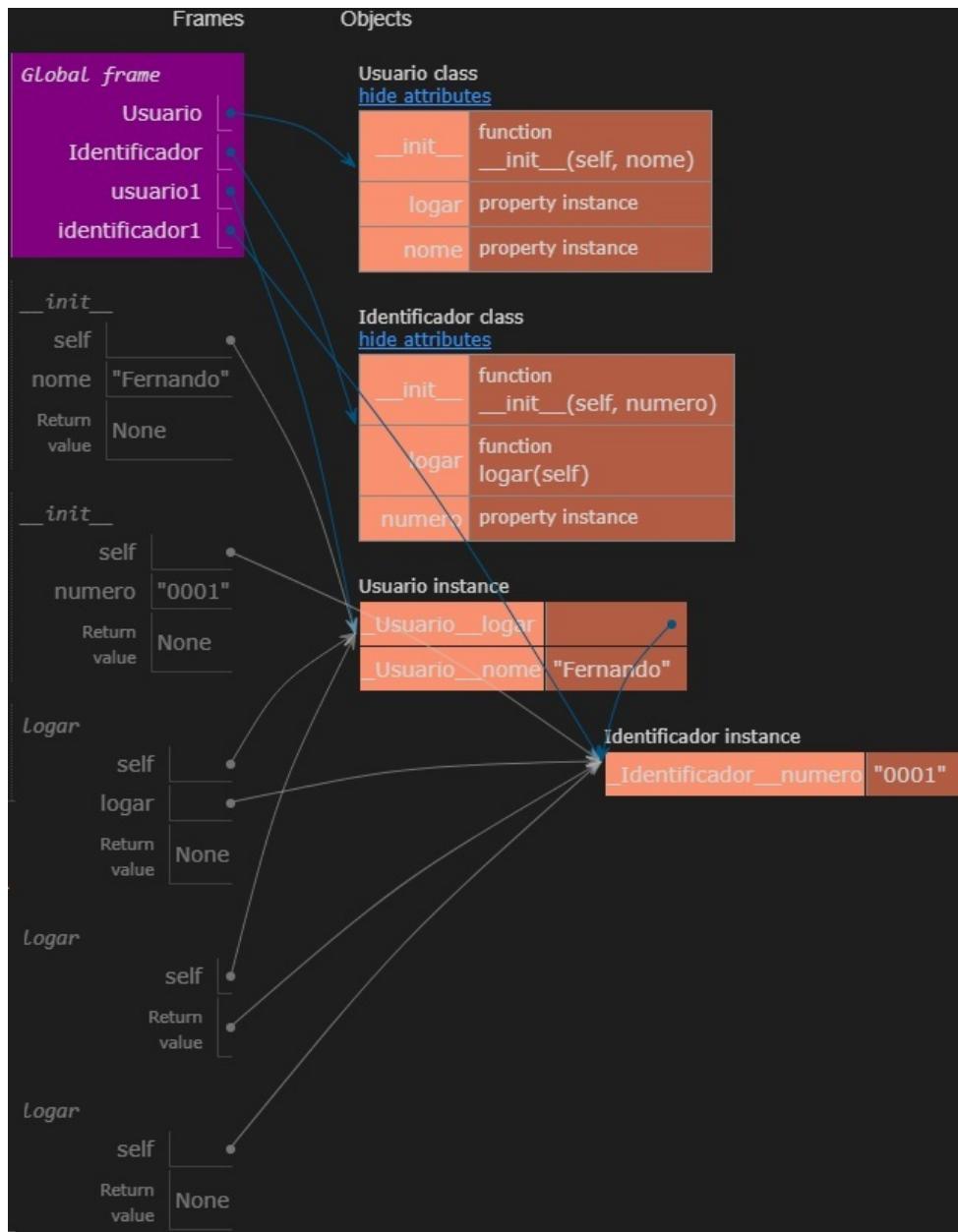
```
    return self._numero
def logar(self):
    print('Logando no sistema...')
```

Na sequência criamos uma nova classe de nome **Identificador** que aqui, apenas como exemplo, seria o gatilho para um sistema de autenticação de usuário em um determinado sistema. Logo, a mesma também possui um método construtor que recebe um número para identificação do usuário, novamente, é criado um **getter** e um **setter** para tornar o núcleo desse código instanciável.

```
usuario1 = Usuario('Fernando')
identificador1 = Identificador('0001')

usuario1.logar = identificador1
usuario1.logar.logar()
```

Por fim, é criado um objeto no corpo de nosso código de nome **usuario1** que instancia a classe **Usuario**, lhe passando como atributo '**Fernando**'. O mesmo é feito para o objeto **identificador1**, que instancia a classe **Identificador** passando como parâmetro '**0001**'. Agora vem a parte onde estaremos de fato associando essas classes e fazendo as mesmas trabalhar em conjunto. Para isso por meio de **usuario.logar = identificador** associamos **0001** a **Fernando**, e supondo que essa é a validação necessária para esse sistema de identificação (ou pelo menos uma das etapas), por meio da função **usuario1.logar.logar()** o retorno é: **Logando no sistema...**



Raciocine que este exemplo básico é apenas para entender a lógica de associação, obviamente um sistema de autenticação iria exigir mais dados como senha, etc... assim como confrontar esses dados com um banco de dados... Também é importante salientar que quanto mais associações houver em nosso código, de forma geral mais eficiente é a execução do mesmo ao invés de blocos e mais blocos de códigos independentes e repetidos para cada situação, porém, quanto mais robusto for o código, maior também será a suscetibilidade a erro, sendo assim, é interessante testar o código a cada bloco criado para verificar se não há nenhum erro de lógica ou sintaxe.

Agregação e composição de classes

Começando o entendimento pela sua lógica, anteriormente vimos o que basicamente é a associação de classes, onde temos mais de uma classe, elas podem compartilhar métodos e atributos entre si ao serem instanciadas, porém uma pode ser executada independentemente da outra. Já quando falamos em agregação e composição, o laço entre essas classes e suas estruturas passa a ser tão forte que uma depende da outra. Raciocine que quando estamos realizando a composição de uma classe, uma classe tomará posse dos objetos das outras classes, de modo que se algo corromper essa classe principal, as outras param de funcionar também.

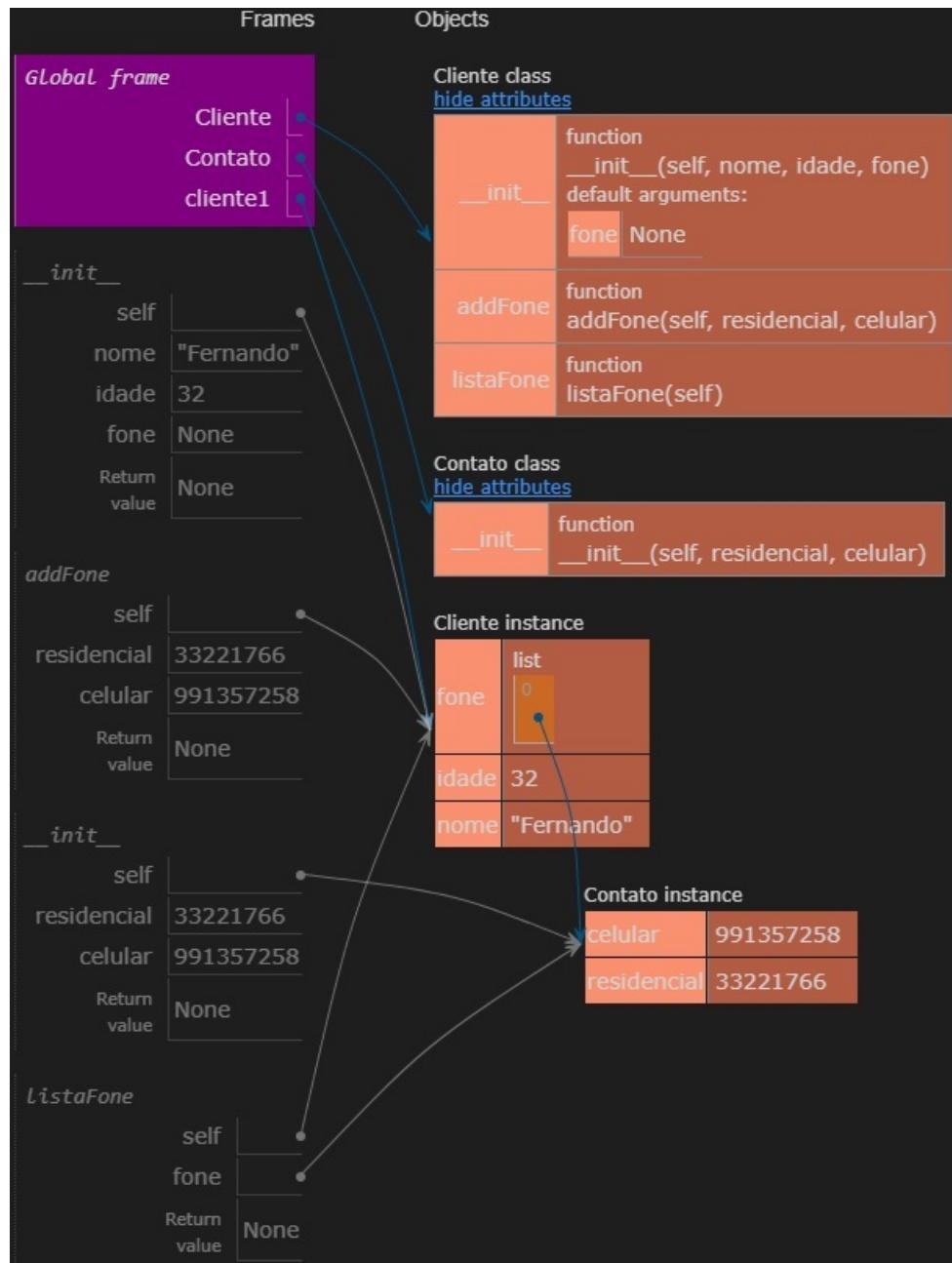
```
class Cliente:  
    def __init__(self, nome, idade, fone=None):  
        self.nome = nome  
        self.idade = idade  
        self.fone = []  
  
    def addFone(self, residencial, celular):  
        self.fone.append(Contato(residencial, celular))  
    def listaFone(self):  
        for fone in self.fone:  
            print(fone.residencial, fone.celular)
```

Para esse exemplo, inicialmente criamos uma classe de nome **Cliente**, ela possui seu método construtor que recebe um **nome**, uma **idade** e um ou mais telefones para contato, isto porque como objeto de classe **fone** inicialmente recebe uma lista em branco. Em seguida é criado um método de classe responsável por alimentar essa lista com números de telefone. Porém, repare no diferencial, em **self.fone**, estamos adicionando dados instanciando como parâmetro outra classe, nesse caso a classe **Contato** com toda sua estrutura. Como dito anteriormente, este laço, instanciando uma classe dentro de outra classe é um laço forte onde, se houver qualquer exceção no código, tudo irá parar de funcionar uma vez que tudo está interligado operando em conjunto.

```
cliente1 = Cliente('Fernando', 32)  
cliente1.addFone(33221766, 991357258)  
print(cliente1.nome)  
print(cliente1.listaFone())
```

Seguindo com o exemplo, é criada uma variável de nome **cliente1** que instancia **Cliente** passando como atributos de classe **Fernando**, **32**. Em seguida é chamada a função **addFone()** passando como parâmetros **33221766** e **991357258**. Por fim, por meio do comando **print()** pedimos que sejam exibidos os dados de **cliente1.nome** de **cliente1.listaFone()**. Nesse caso o retorno será:

Fernando
33221766 991357258



Herança Simples

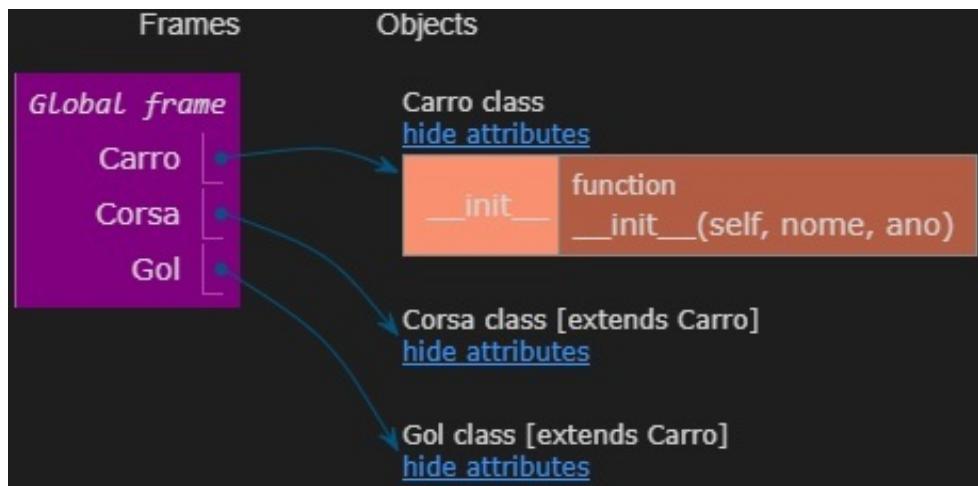
Na estrutura de código de um determinado programa todo orientado a objetos, é bastante comum que algumas classes em teoria possuam exatamente as mesmas características que outras, porém isso não é nada eficiente no que diz respeito a quantidade de linhas de código uma vez que esses blocos, já que são idênticos, estão repetidos no código. Por Exemplo:

```
class Corsa:  
    def __init__(self, nome, ano):  
        self.nome = nome  
        self.ano = ano  
  
class Gol:  
    def __init__(self, nome, ano):  
        self.nome = nome  
        self.ano = ano
```

Dado o exemplo acima, note que temos duas classes para dois carros diferentes, **Corsa** e **Gol**, e repare que as características que irão identificar os mesmos, aqui nesse exemplo bastante básico, seria um **nome** e o **ano** do veículo. Duas classes com a mesma estrutura de código repetida duas vezes. Podemos otimizar esse processo criando uma classe molde de onde serão lidas e interpretadas todas essas características de forma otimizada por nosso interpretador. Vale lembrar que neste exemplo são apenas duas classes, não impactando na performance, mas numa aplicação real pode haver dezenas ou centenas de classes idênticas em estrutura, o que acarreta em lentidão se lidas uma a uma de forma léxica.

```
class Carro:  
    def __init__(self, nome, ano):  
        self.nome = nome  
        self.ano = ano  
  
class Corsa(Carro):  
    pass  
  
class Gol(Carro):  
    pass
```

Reformulando o código, é criada uma classe de nome **Carro** que servirá de molde para as outras, dentro de si ela tem um método construtor assim como recebe como atributos de classe um **nome** e um **ano**. A partir de agora as classes **Corsa** e **Gol** simplesmente recebem como parâmetro a classe **Carro**, recebendo toda sua estrutura. Internamente o interpretador não fará distinção e assumirá que cada classe é uma classe normal, independente, inclusive com alocações de memória separadas para cada classe, porém a estrutura do código dessa forma é muito mais enxuta.



Na literatura, principalmente autores que escreveram por versões anteriores do Python costumavam chamar esta estrutura de classes e subclasses, apenas por convenção, para que seja facilitada a visualização da hierarquia das mesmas.

Vale lembrar também que uma “subclasse” pode ter mais métodos e atributos particulares a si, sem problema nenhum, a herança ocorre normalmente a tudo o que estiver codificado na classe mãe, porém as classes filhas desta tem flexibilidade para ter mais atributos e funções conforme necessidade. Ainda sob o exemplo anterior, a subclasse Corsa, por exemplo, poderia ser mais especializada tendo maiores atributos dentro de si além do nome e ano herdado de Carro.

Cadeia de heranças

Como visto no tópico anterior, uma classe pode herdar outra sem problema nenhum, desde que a lógica estrutural e sintática esteja correta na hora de definir as mesmas. Extrapolando um pouco, vale salientar que essa herança pode ocorrer em diversos níveis, na verdade, não há um limite para isto, porém é preciso tomar bastante cuidado para não herdar características desnecessárias que possam tornar o código ineficiente. Ex:

```
class Carro:  
    def __init__(self, nome, ano):  
        self.nome = nome  
        self.ano = ano  
  
class Gasolina(Carro):  
    def __init__(self, tipogasolina=True, tipoalcool=False):  
        self.tipogasolina = tipogasolina  
        self.tipoalcool = tipoalcool  
  
class Jeep(Gasolina):  
    pass
```

Repare que nesse exemplo inicialmente é criada uma classe **Carro**, dentro de si ela possui um método construtor onde é repassado como atributo de classe um **nome** e um **ano**. Em seguida é criada uma classe de nome **Gasolina** que herda toda estrutura de **Carro** e dentro de si define que o veículo desta categoria será do tipo gasolina. Por fim é criada uma última classe se referindo a um carro marca **Jeep**, que herda toda estrutura de **Gasolina** e de **Carro** pela hierarquia entre classes. Seria o mesmo que:

```
class Jeep:  
    def carro():  
        def __init__(self, nome, ano):  
            self.nome = nome  
            self.ano = ano  
  
        def gasolina(self, tipogasolina=True, tipoalcool=False):  
            self.tipogasolina = tipogasolina  
            self.tipoalcool = tipoalcool  
  
    jeep = Jeep()
```

Código usual, porém, ineficiente. Assim como estático, de modo que quando houvesse a criação de outros veículos não seria possível reaproveitar nada de dentro dessa Classe a não ser por instâncias de fora da mesma, o que gera muitos caminhos de leitura para o interpretador, tornando a performance do código ruim.

Lembrando que a vantagem pontual que temos ao realizar heranças entre classes é a reutilização de seus componentes internos no lugar de blocos e mais blocos de código repetidos.

```
class Carro:
```

```

def __init__(self, nome, ano):
    self.nome = nome
    self.ano = ano

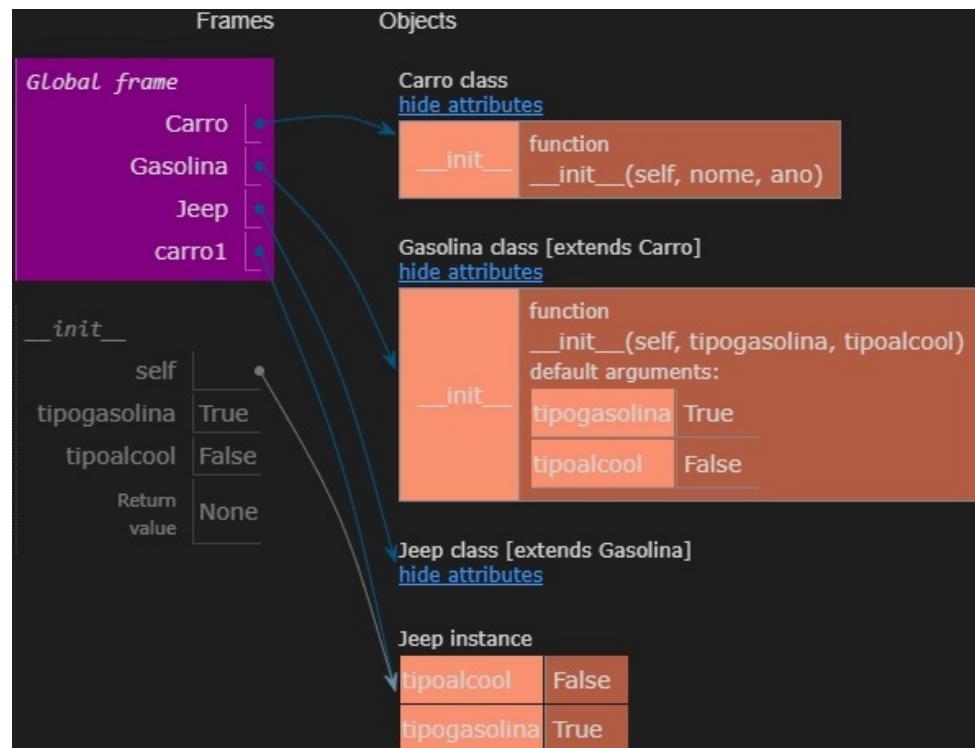
class Gasolina(Carro):
    def __init__(self, tipogasolina=True, tipoalcool=False):
        self.tipogasolina = tipogasolina
        self.tipoalcool = tipoalcool

class Jeep(Gasolina):
    pass

carro1 = Jeep()
print(carro1.tipogasolina)

```

Código otimizado via herança e cadeia de hierarquia simples. Executando nossa função `print()` nesse caso o retorno será: **True**



Herança Múltipla

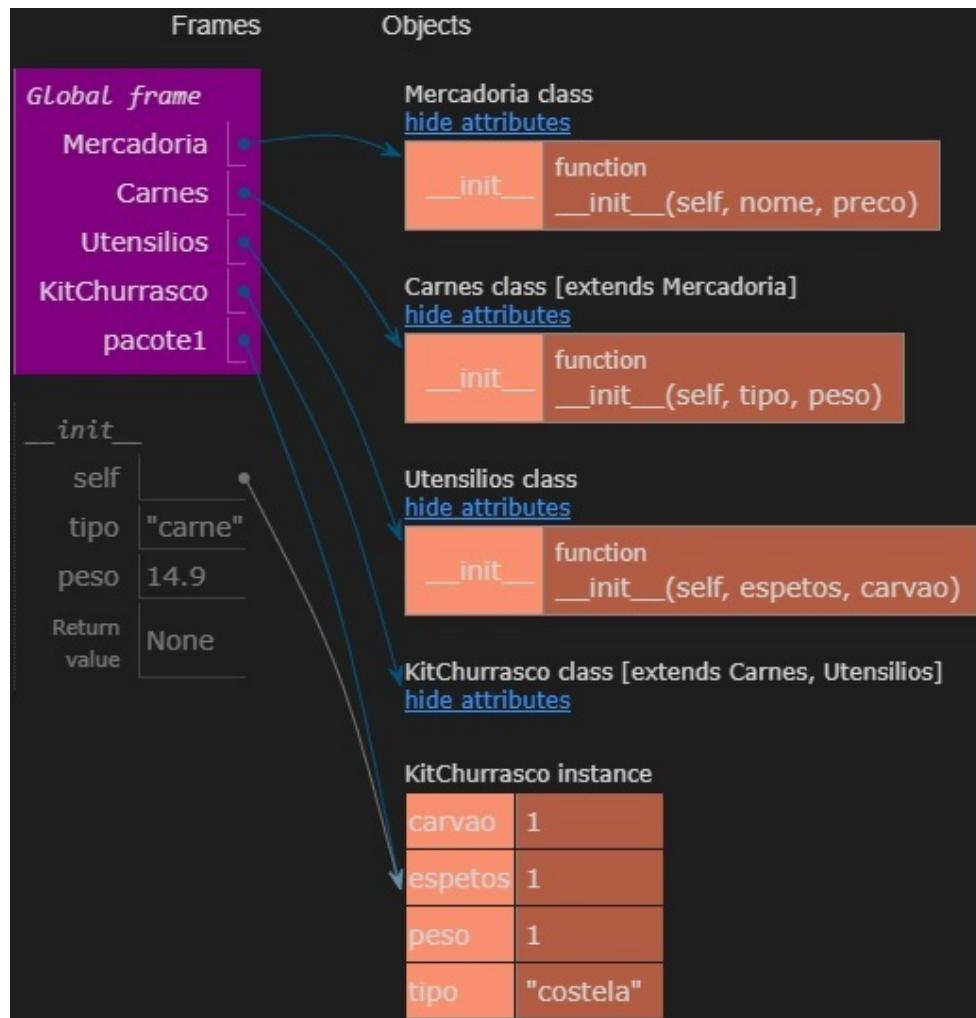
Entendida a lógica de como uma classe herda toda a estrutura de outra classe, podemos avançar um pouco mais fazendo o que é comumente chamado de herança múltipla. Raciocine que numa herança simples, criávamos uma determinada classe assim como todos seus métodos e atributos de classe, de forma que podíamos instanciar / passar ela como parâmetro de uma outra classe. Respeitando a sintaxe, é possível realizar essa herança vinda de diferentes classes, importando dessa forma seus conteúdos. Por exemplo:

```
class Mercadoria:  
    def __init__(self, nome, preco):  
        self.nome = nome  
        self.preco = preco  
  
class Carnes(Mercadoria):  
    def __init__(self, tipo, peso):  
        self.tipo = tipo  
        self.peso = peso  
  
class Utensilios:  
    def __init__(self, espertos, carvao):  
        self.espetos = espertos  
        self.carvao = carvao  
  
class KitChurrasco(Carnes, Utensilios):  
    pass
```

Para este exemplo, vamos supor que esse seria o background de um simples sistema de inventário de um mercado, onde as categorias de produtos possuem estrutura em classes distintas, porém que podem ser combinadas para montar “kits” de produtos. Então note que inicialmente é criada uma classe de nome **Mercadoria**, que por sua vez possui um método construtor assim como **nome** e **preco** das mercadorias como atributos de classe. Em seguida é criada uma classe de nome **Carnes** que herda toda estrutura de **Mercadoria** e adiciona os atributos de classe referentes ao **tipo** de carne e seu **peso**. Na sequência é criada uma classe de nome **Utensilios** que não herda nada das classes anteriores, e dentro de si possui estrutura de método construtor assim como seus atributos de classe. Por fim, é criada uma classe de nome **KitChurrasco**, que por sua vez herda **Carnes** (que herda **Mercadoria**) e **Utensilios** com todo seu conteúdo.

```
pacote1 = KitChurrasco('carne', 14.90)  
pacote1.tipo = 'costela'  
pacote1.peso = 1  
pacote1.espetos = 1  
pacote1.carvao = 1
```

A partir disso, como nos exemplos anteriores, é possível instanciar objetos e definir dados/valores para os mesmos normalmente, uma vez que **KitChurrasco** agora, devido as características que herdou, possui campos para **nome**, **preco**, **tipo**, **peso**, **espertos** e **carvao** dentro de si.



Sobreposição de membros

Um dos cuidados que devemos ter ao manipular nossas classes em suas hierarquias e heranças é o fato de apenas realizarmos sobreposições quando de fato queremos alterar/sobrescrever um determinado dado/valor/método dentro da mesma. Em outras palavras, anteriormente vimos que o interpretador do Python realiza a chamada leitura léxica do código, ou seja, ele lê linha por linha (de cima para baixo) e linha por linha (da esquerda para direita), dessa forma, podemos reprogramar alguma linha ou bloco de código para que na sequência de sua leitura/interpretação o código seja alterado. Por exemplo:

```
class Pessoa:  
    def __init__(self, nome):  
        self.nome = nome  
  
    def Acao1(self):  
        print(f'{self.nome} está dormindo')  
  
class Jogador1(Pessoa):  
    def Acao2(self):  
        print(f'{self.nome} está comendo')  
  
class SaveJogador1(Jogador1):  
    pass
```

Inicialmente criamos a classe **Pessoa**, dentro de si um método construtor que recebe um nome como atributo de classe. Também é definida uma **Acao1** que por sua vez por meio da função **print()** exibe uma mensagem. Em seguida é criada uma classe de nome **Jogador1** que herda tudo de **Pessoa** e por sua vez, apenas tem um método **Acao2** que exibe também uma determinada mensagem. Por fim é criado uma classe **SaveJogador1** que herda tudo de **Jogador1** e de **Pessoa**.

```
p1 = SaveJogador1('Fernando')  
print(p1.nome)
```

Trabalhando sobre essa cadeia de classes, criamos uma variável de nome **p1** que instancia **SaveJogador1** e passa como parâmetro '**Fernando**', pela hierarquia destas classes, esse parâmetro alimentará **self.nome** de **Pessoa**. Por meio da função **print()** passando como parâmetros **p1.nome** o retorno é: **Fernando**

```
p1.Acao1()  
p1.Acao2()
```

Da mesma forma, instanciando qualquer coisa de dentro de qualquer classe da hierarquia, se não houver nenhum erro de sintaxe tudo será executado normalmente.

Neste caso o retorno será:

Fernando está dormindo
Fernando está comendo

```
class Pessoa:  
    def __init__(self, nome):
```

```

self.nome = nome

def Acao1(self):
    print(f'{self.nome} está dormindo')

class Jogador1(Pessoa):
    def Acao2(self):
        print(f'{self.nome} está comendo')

class SaveJogador1(Jogador1):
    def Acao1(self):
        print(f'{self.nome} está acordado')

```

Por fim, partindo para uma sobreposição de fato, note que agora em **SaveJogador1** criamos um método de classe de nome **Acao1**, dentro de si uma função para exibir uma determinada mensagem. Repare que **Acao1** já existia em **Pessoa**, mas o que ocorre aqui é que pela cadeia de heranças **SaveJogador1** criando um método **Acao1** irá sobrepor esse método já criado anteriormente. Em outras palavras, dada a hierarquia entre essas classes, o interpretador irá considerar pela sua leitura léxica a última alteração das mesmas, **Acao1** presente em **SaveJogador1** é a última modificação desse método de classe, logo, a função interna do mesmo irá ser executada no lugar de **Acao1** de **Pessoa**, que será ignorada.

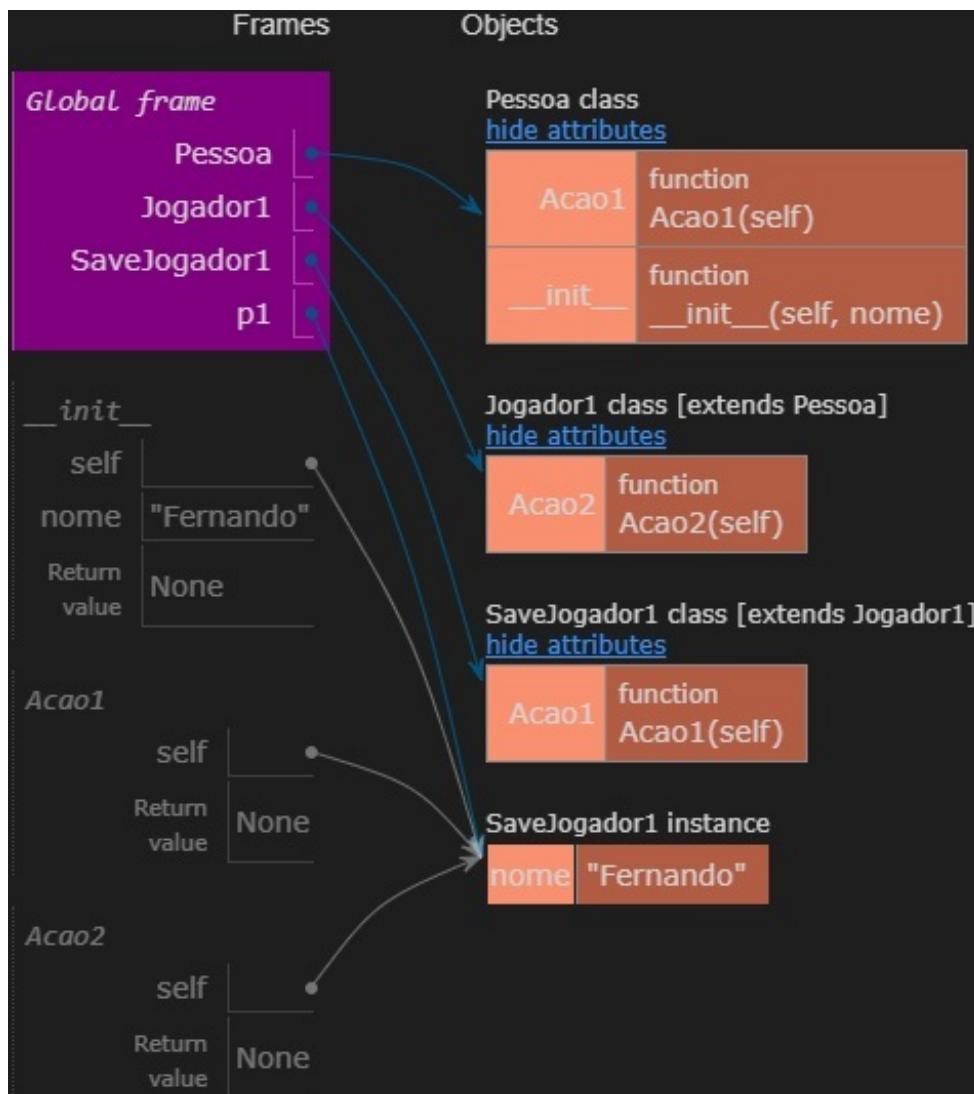
```

p1 = SaveJogador1('Fernando')
p1.Acao1()
p1.Acao2()

```

Nesse caso o retorno será:

Fernando está acordado (a última instrução atribuída a **Acao1**)
Fernando está comendo



Outra funcionalidade que eventualmente pode ser utilizada é a de, justamente executar uma determinada ação da classe mãe e posteriormente a sobrescrever, isso é possível por meio da função reservada `super()`. Esta função em orientação a objetos faz com que seja respeitada a hierarquia de classe mãe / super classe em primeiro lugar e posteriormente as classes filhas / sub classes.

```
class SaveJogador1(Jogador1):
    def Acao1(self):
        super().Acao1()
        print(f'{self.nome} está acordado')
```

Neste caso, primeiro será executada a **Acao1** de **Pessoa** e em seguida ela será sobreescrita por **Acao1** de **SaveJogador1**. Nesse caso o retorno será:

Fernando está dormindo (Acao1 de Pessoa)
Fernando está acordado (Acao1 de SaveJogador1)
Fernando está comendo

Avançando com nossa linha de raciocínio, ao se trabalhar com heranças deve-se tomar muito cuidado com as hierarquias entre as classes dentro de sua ordem de leitura léxica. Raciocine que o mesmo padrão usado no exemplo anterior vale para qualquer coisa, até mesmo um método construtor pode ser sobreescrito, logo, devemos tomar os devidos cuidados na hora de usar os mesmos.

Também é importante lembrar que por parte de sobreposições, quando se trata de heranças múltiplas, a ordem como as classes são passadas como parâmetro influência nos resultados. Ex:

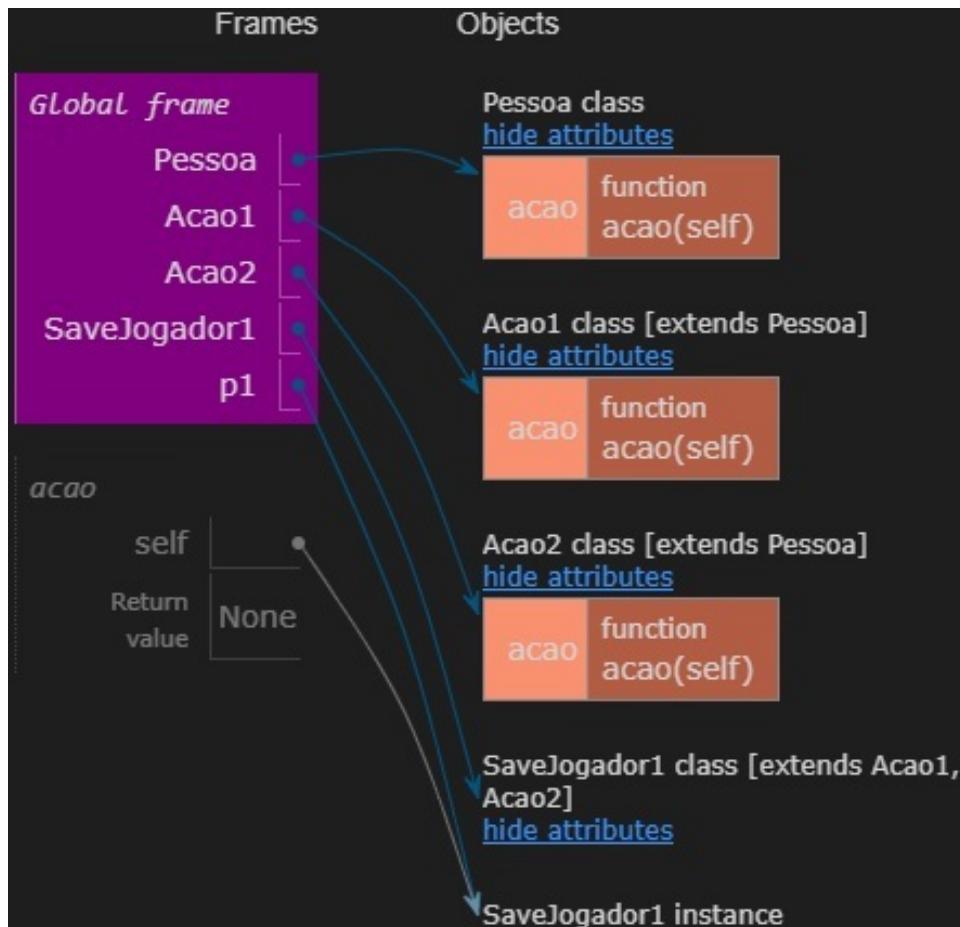
```
class Pessoa:  
    def acao(self):  
        print('Inicializando o sistema')  
  
class Acao1(Pessoa):  
    def acao(self):  
        print('Sistema pronto para uso')  
  
class Acao2(Pessoa):  
    def acao(self):  
        print('Desligando o sistema')  
  
class SaveJogador1(Acao1, Acao2):  
    pass  
  
p1 = SaveJogador1()  
p1.acao()
```

Apenas como exemplo, repare que foram criadas 4 classes, **Pessoa**, **Acao1**, **Acao2** e **SaveJogador1** respectivamente, dentro de si apenas existe um método de classe designado a exibir uma mensagem. Para esse exemplo, o que importa é a forma como **SaveJogador1** está herdando características de **Acao1** e **Acao2**, como dito anteriormente, essa ordem ao qual as classes são passadas como parâmetro aqui influenciará os resultados da execução do código.

Nesse exemplo, **SaveJogador1(Acao1, Acao2)**, o retorno será: **Sistema pronto para uso**

```
class SaveJogador1(Acao2, Acao1):  
    pass
```

Exatamente a mesma estrutura anterior, porém com ordem inversa, **SaveJogador1(Acao2, Acao1)**, o retorno será: **Desligando o sistema**



Apenas fazendo um adendo, caso estes tópicos ainda gerem alguma dúvida para você.
Uma forma de diferenciarmos a forma como tratamos as interações entre classes dentro do Python é a seguinte:

Associação: Se dá quando uma classe usa outro ou algum objeto de outra.

Aggregação: Se dá quando um ou mais objetos de classe é compartilhado, usado por duas ou mais classes (para que não seja necessário programar esse atributo para cada uma delas).

Composição: Se dá quando uma classe é dona de outra pela interligação de seus atributos e a forma sequencial como uns dependem dos outros.

Herença: Se dá quando estruturalmente um objeto é outro objeto, no sentido de que ele literalmente herda todas características e funcionalidades do outro, para que o código fique mais enxuto.

Classes abstratas

Se você chegou até este tópico do livro certamente não terá nenhum problema para entender este modelo de classe, uma vez que já o utilizamos implicitamente em outros exemplos anteriores. Mas o fato é que, como dito lá no início, para alguns autores uma classe seria um molde a ser utilizado para se guardar todo e qualquer tipo de dado associado a um objeto de forma que possa ser utilizado livremente de forma eficiente ao longo do código. Sendo assim, quando criamos uma classe vazia, ou bastante básica, que serve como base estrutural para outras classes, já estamos trabalhando com uma classe abstrata. O que faremos agora é definir manualmente este tipo de classe, de forma que sua forma estrutural irá forçar as classes subsequentes a criar suas especializações a partir dela. Pode parecer um tanto confuso, mas com o exemplo tudo ficará mais claro.

Como dito anteriormente, na verdade já trabalhamos com este tipo de classe, mas de forma implícita e assim o interpretador do Python não gerava nenhuma barreira quanto a execução de uma classe quando a mesma era abstrata. Agora, usaremos para este exemplo, um módulo do Python que nos obrigará a abstrair manualmente as classes a partir de uma sintaxe própria.

Sendo assim, inicialmente precisamos importar os módulos **ABC** e **abstractmethod** da biblioteca **abc**.

```
from abc import ABC, abstractclassmethod
```

Realizadas as devidas importações, podemos prosseguir com o exemplo:

```
class Pessoa(ABC):
    @abstractclassmethod
    def logar(self):
        pass

class Usuario(Pessoa):
    def logar(self):
        print('Usuario logado no sistema')
```

Inicialmente criamos uma classe **Pessoa**, que já herda **ABC** em sua declaração, dentro de si, note que é criado um decorador **@abstractclassmethod**, que por sua vez irá sinalizar ao interpretador que todos os blocos de código que vierem na sequência dessa classe, devem ser sobreescritas em suas respectivas classes filhas dessa hierarquia. Em outras palavras, **Pessoa** no momento dessa declaração, possui um método chamado **logar** que obrigatoriamente deverá ser criado em uma nova classe herdeira, para que possa ser instanciada e realizar suas devidas funções. Dando sequência, repare que em seguida criamos uma classe **Usuario**, que herda **Pessoa**, e dentro de si cria o método **logar** para sobrepor/sobrescrever o método **logar** de **Pessoa**. Este, por sua vez, exibe uma mensagem pré-definida por meio da função **print()**.

```
user1 = Pessoa()
user1.logar()
```

A partir do momento que **Pessoa** é identificada como uma classe abstrata por nosso decorador, a mesma passa a ser literalmente apenas um molde. Seguindo com o exemplo, tentando fazer a

associação que estamos acostumados a fazer, atribuindo essa classe a um objeto qualquer, ao tentar executar a mesma será gerado um erro de interpretação.

Traceback (most recent call last):

File "c:/Users/Fernando/Documents/001.py", line 12, in <module>

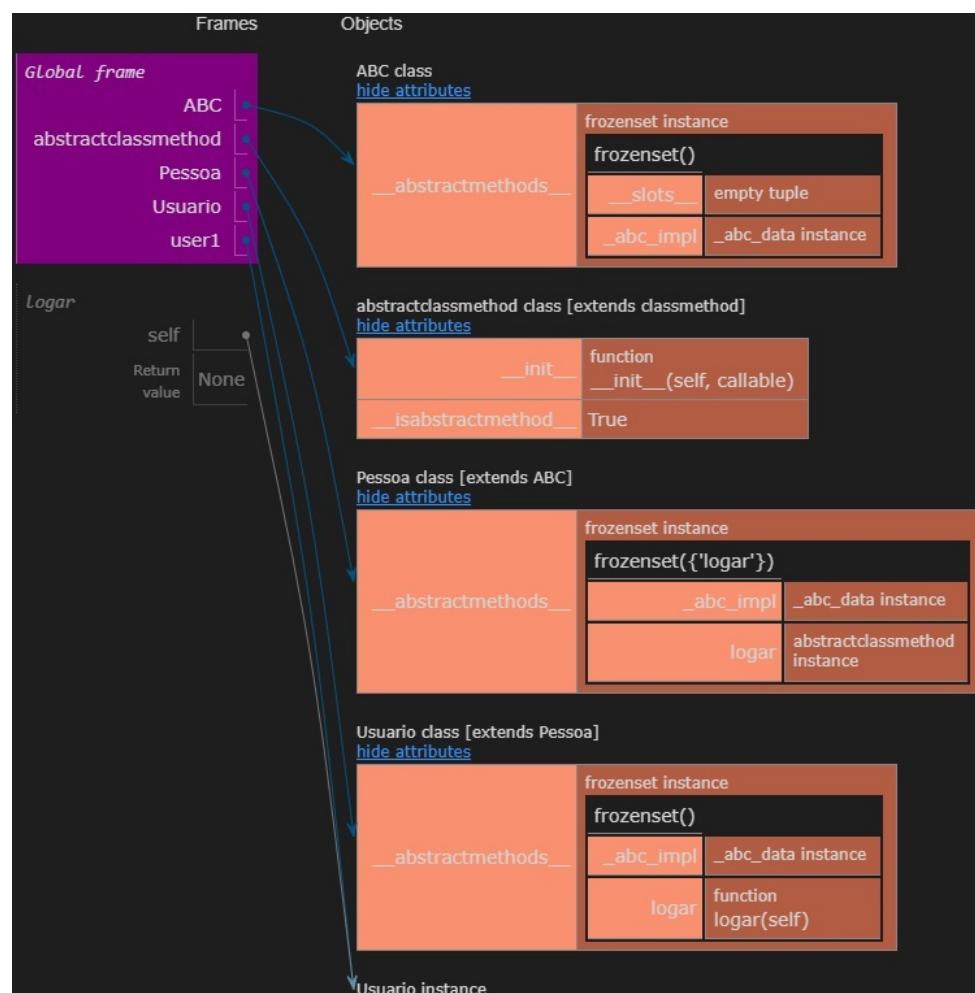
 user1 = Pessoa()

TypeError: Can't instantiate abstract class Pessoa with abstract methods logar

Em tradução livre: Não é possível inicializar a classe abstrata Pessoa com o método abstrato logar.

```
user1 = Usuario()
user1.logar()
```

Porém instanciando **Usuario**, que herda **Pessoa**, ao tentar executar o método de classe logar que consta no corpo do mesmo, este funciona perfeitamente. Nesse caso o retorno será: **Usuario logado no sistema**



Sendo assim, o que precisa ficar bem claro, inicialmente, é que uma das formas que temos de proteger uma classe que apenas nos serve de molde, para que essa seja herdada, mas não sobrescrita, é definindo a mesma manualmente como uma classe abstrata.

Polimorfismo

Avançando mais um pouco com nossos estudos, hora de abordar um princípio da programação orientada a objetos chamada polimorfismo. Como o próprio nome já sugere, polimorfismo significa que algo possui muitas formas (ou ao menos mais que duas) em sua estrutura. Raciocine que é perfeitamente possível termos classes derivadas de uma determinada classe mãe / super classe que possuem métodos iguais, porém comportamentos diferentes dependendo sua aplicação no código.

Alguns autores costumam chamar esta característica de classe de “assinatura”, quando se refere a mesma quantidade e tipos de parâmetros, porém com fins diferentes. Último ponto a destacar antes de partirmos para o código é que, polimorfismo normalmente está fortemente ligado a classes abstratas, sendo assim, é de suma importância que a lógica deste tópico visto no capítulo anterior esteja bem clara e entendida, caso contrário poderá haver confusão na hora de identificar tais estruturas de código para criar suas devidas associações.

Entenda que uma classe abstrata, em polimorfismo, obrigatoriamente será um molde para criação das demais classes onde será obrigatório realizar a sobreposição do(s) método(s) de classe que a classe mãe possuir.

```
from abc import ABC, abstractclassmethod

class Pessoa(ABC):
    @abstractclassmethod
    def logar(self, chavesegurança):
        pass

class Usuario(Pessoa):
    def logar(self, chavesegurança):
        print('Usuario logado no sistema')

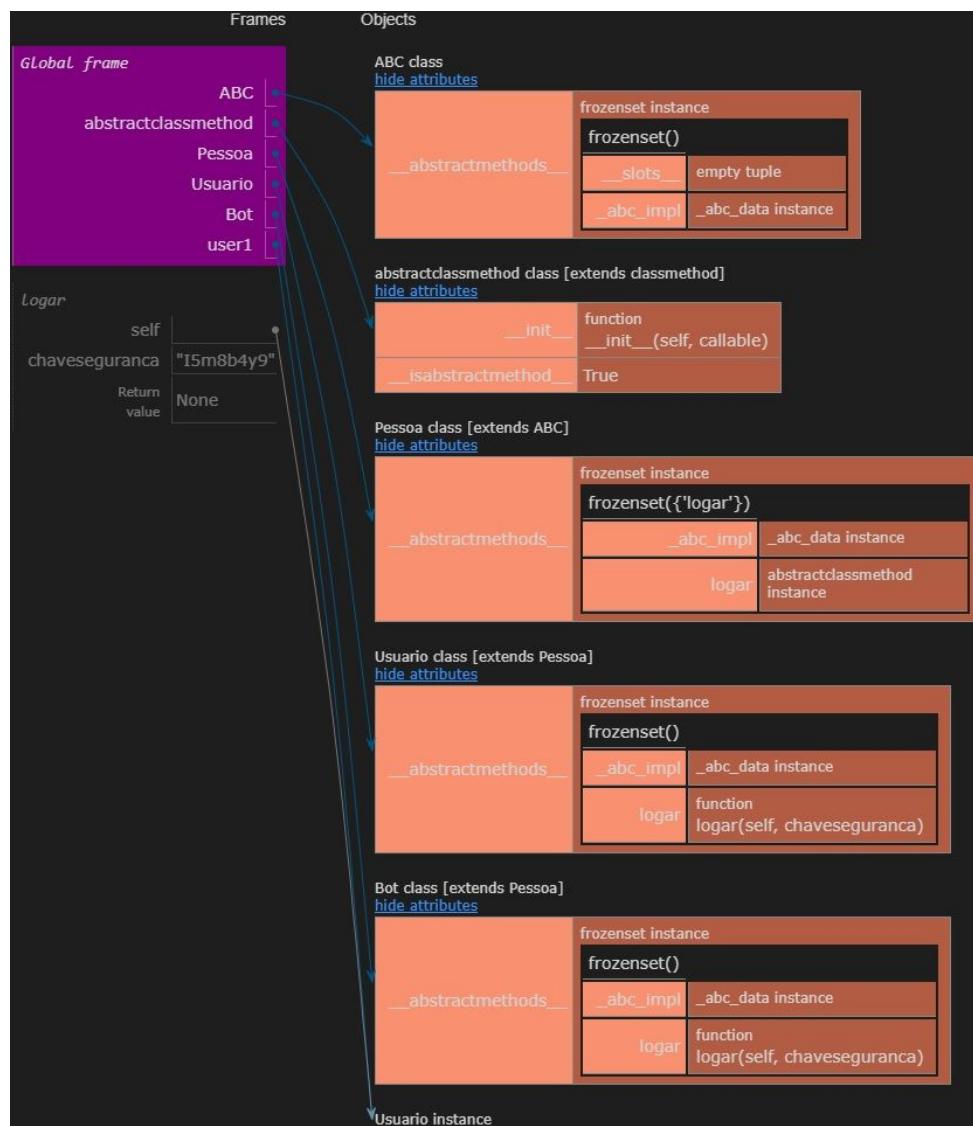
class Bot(Pessoa):
    def logar(self, chavesegurança):
        print('Sistema rodando em segundo plano')
```

Seguindo com um exemplo muito próximo ao anterior, apenas para melhor identificação, note que inicialmente são feitas as importações dos módulos **ABC** e **abstractclassmethod** da biblioteca **abc**. Em seguida é criada uma classe de nome **Pessoa** que herda **ABC**, dentro de si existe um decorador e um método de classe que a caracteriza como uma classe abstrata. Em outras palavras, toda classe que herdar a estrutura de **Pessoa** terá de redefinir o método **logar**, fornecendo uma **chavedesegurança** que aqui não está associada a nada, mas na prática seria parte de um sistema de login. Na sequência é criada a classe **Usuário** que herda **Pessoa**, e pela sua estrutura, repare que ela é polimórfica, possuindo a mesma assinatura de objetos instanciados ao método **logar**, porém há uma função personalizada que está programada para exibir uma mensagem. O mesmo é feito com uma terceira classe chamada **Bot** que herda **Pessoa**, que por sua vez herda **ABC**, tudo sob a mesma “assinatura”.

```
user1 = Usuario()
```

```
user1.logar('I5m8b4y9')
```

Instanciando **Usuario** a um objeto qualquer e aplicando sobre si parâmetros (neste caso, fornecendo uma senha para **chavesseguranca**), ocorrerá a execução da respectiva função. Neste caso o retorno será: **Usuario logado no sistema**



Sobrecarga de operadores

Quando damos nossos primeiros passos no aprendizado do Python, em sua forma básica, programando de forma estruturada, um dos primeiros tópicos que nos é apresentado são os operadores. Estes por sua vez são símbolos reservados ao sistema que servem para fazer operações lógicas ou aritméticas, de forma que não precisamos programar do zero tais ações, pela sintaxe, basta usar o operador com os tipos de dados aos quais queremos a interação e o interpretador fará a leitura dos mesmos normalmente. Em outras palavras, se declaramos duas variáveis com números do tipo inteiro, podemos por exemplo, via operador de soma + realizar a soma destes valores, sem a necessidade de programar do zero o que seria uma função que soma dois valores.

Se tratando da programação orientada a objetos, o que ocorre é que quando criamos uma classe com variáveis/objetos dentro dela, estes dados passam a ser um tipo de dado novo, independente, apenas reconhecido pelo interpretador como um objeto. Para ficar mais claro, raciocine que uma variável normal, com valor atribuído de 4 por exemplo, é automaticamente interpretado por nosso interpretador como int (número inteiro, sem casas decimais), porém a mesma variável, exatamente igual, mas declarada dentro de uma classe, passa a ser apenas um objeto geral para o interpretador.

Números inteiros podem ser somados, subtraídos, multiplicados, elevados a uma determinada potência, etc... um objeto dentro de uma classe não possui essas funcionalidades a não ser que manualmente realizemos a chamada sobrecarga de operadores. Por exemplo:

```
class Caixa:  
    def __init__(self, largura, altura):  
        self.largura = largura  
        self.altura = altura  
  
caixa1 = Caixa(10,10)  
caixa2 = Caixa(10,20)
```

Aqui propositalmente simulando este erro lógico, inicialmente criamos uma classe de nome **Caixa**, dentro dela um método construtor que recebe um valor para largura e um para altura, atribuindo esses valores a objetos internos. No corpo de nosso código, criando variáveis que instanciam nossa classe **Caixa** e atribuem valores para **largura** e **altura**, seria normal raciocinar que seria possível, por exemplo, realizar a soma destes valores.

```
print(caixa1 + caixa2)
```

Porém o que ocorre ao tentar realizar uma simples operação de soma entre tais valores o retorno que temos é um traceback.

TypeError: unsupported operand type(s) for +: 'Caixa' and 'Caixa'

Em tradução livre: Erro de Tipo: tipo de operador não suportado para +: Caixa e Caixa.

Repare que o erro em si é que o operador de soma não foi reconhecido como tal, pois esse símbolo de + está fora de contexto uma vez que estamos trabalhando com orientação a objetos.

Sendo assim, o que teremos de fazer é simplesmente buscar os operadores que sejam reconhecidos e processados neste tipo de programação.

Segue uma lista com os operadores mais utilizados, assim como o seu método correspondente para uso em programação orientada a objetos.

Operador	Método	Operação
+	<u>add</u>	Adição
-	<u>sub</u>	Subtração
*	<u>mul</u>	Multiplicação
/	<u>div</u>	Divisão
//	<u>floordiv</u>	Divisão inteira
%	<u>mod</u>	Módulo
**	<u>pow</u>	Potência
<	<u>lt</u>	Menor que
>	<u>gt</u>	Maior que
<=	<u>le</u>	Menor ou igual a
>=	<u>ge</u>	Maior ou igual a
==	<u>eq</u>	Igual a
!=	<u>ne</u>	Diferente de

Dando continuidade a nosso entendimento, uma vez que descobrimos que para cada tipo de operador lógico/aritmético temos um método de classe correspondente, tudo o que teremos de fazer é de fato criar este método dentro de nossa classe, para forçar o interpretador a reconhecer que, nesse exemplo, é possível realizar a soma dos dados/valores atribuídos aos objetos ali instanciados.

```
class Caixa:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    def __add__(self, other):
        pass
```

Retornando ao código, note que simplesmente é criado um método de classe de nome add que recebe como atributos um valor para si e um valor a ser somado neste caso. add é uma palavra reservada ao sistema, logo, o interpretador sabe sua função e a executará normalmente dentro desse bloco de código.

```
class Caixa:
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura
```

```
def __add__(self, other):
    largura1 = self.largura + other.largura
    altura1 = self.altura + other.altura
    return Caixa(largura1, altura1)

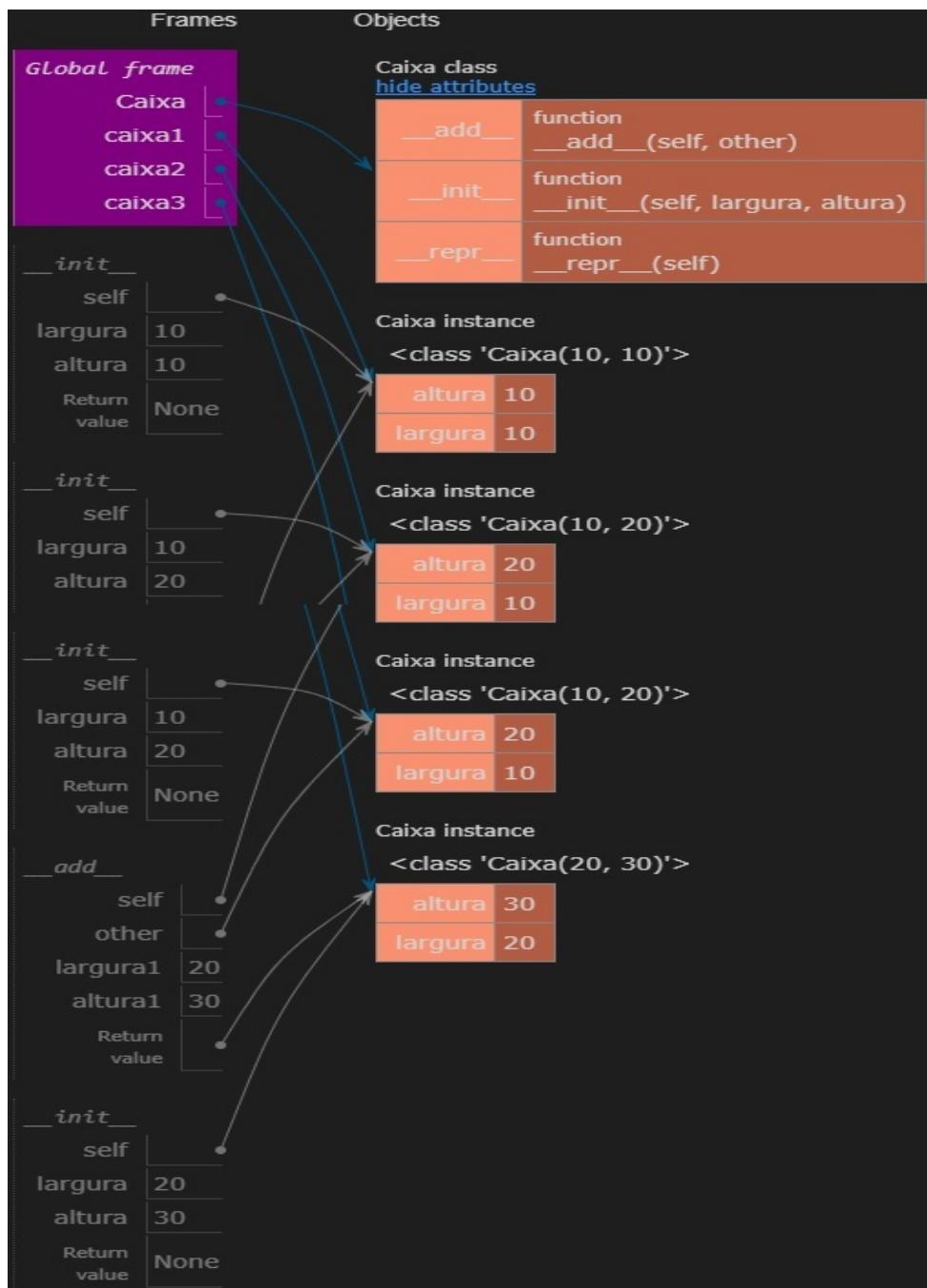
def __repr__(self):
    return f"<class 'Caixa({self.largura}, {self.altura})'>"
```

Dessa forma, podemos criar normalmente uma função de soma personalizada, note que no método `__add__` foi criada uma variável de nome `largura1` que guardará o valor da soma entre as larguras dos objetos. Da mesma forma, `altura1` receberá o valor da soma das alturas, tudo isto retornando para a classe tais valores. Na sequência, apenas para facilitar a visualização, foi criada um método de classe `__repr__` (também reservado ao sistema) que irá retornar os valores das somas explicitamente ao usuário (lembre-se de que como estamos trabalhando dentro de uma classe, por padrão esta soma seria uma mecanismo interno, se você tentar printar esses valores sem esta última função, você terá como retorno apenas o objeto, e não seu valor).

```
caixa1 = Caixa(10,10)
caixa2 = Caixa(10,20)
caixa3 = caixa1 + caixa2

print(caixa3)
```

Por fim, agora instanciando a classe, atribuindo seus respectivos valores, finalmente é possível realizar normalmente a operação de soma entre as variáveis que instanciam essa classe. Nesse exemplo, criando uma variável de nome `caixa3` que recebe como atributo a soma dos valores de `largura` e `altura` de `caixa1` e `caixa2`, o retorno será: `<class 'Caixa(20, 30)'>`



Encerrando, apenas tenha em mente que, de acordo com a funcionalidade que quer usar em seu código, deve procurar o método equivalente e o declarar dentro da classe como um método de classe qualquer, para “substituir” o operador lógico/aritmético de forma a ser lido e interpretador por nosso interpretador.

Tratando exceções

Tanto na programação comum estruturada quanto na orientada a objetos, uma prática comum é, na medida do possível, tentar prever os erros que o usuário pode cometer ao interagir com nosso programa e tratar isso como exceções, de forma que o programa não trave, nem feche inesperadamente, mas acuse o erro que o usuário está cometendo para que o mesmo não o repita, ou ao menos, fique ciente que aquela funcionalidade não está disponível. Na programação estruturada temos aquela estrutura básica de declarar uma exceção e por meio dela, via **try** e **except**, tentar fazer com que o interpretador redirecione as tentativas sem sucesso de executar uma determinada ação para alguma mensagem de erro ao usuário ou para alguma alternativa que retorne o programa ao seu estado inicial. Na programação orientada a objetos esta lógica e sintaxe pode ser perfeitamente usada da mesma forma. Ex:

```
class Erro001(Exception):
    pass

def erro():
    raise Erro001('Ação não permitida!')

try:
    erro()
except Erro001 as msgerro:
    print(f'ERRO: {msgerro}')
```

Repare na estrutura, é normalmente suportado pelo Python, que exceções sejam criadas também como classe. Inicialmente criamos uma classe de nome **Erro001** que herda a função reservada ao sistema **Exception**. A partir disto, criamos uma função que instancia uma variável para **Erro001** atribuindo uma **string** onde consta uma mensagem de erro pré-programada ao usuário (lembrando que a ideia é justamente criar mensagens de erro que orientem o usuário sobre o mesmo, e não as mensagens padrão de erro que o interpretador gera em função de lógica ou sintaxe errada). Em seguida, exatamente igual ao modelo estruturado, criamos as funções **try** e **except**, onde realizamos as devidas associações para que caso o usuário cometa um determinado erro seja chamada a mensagem de exceção personalizada. Neste caso o retorno será: **ERRO: Ação não permitida!**

Capítulo Final

Tudo o que tem um começo... tem um fim, e assim encerramos este pequeno livro. Espero que ao chegar nestas linhas finais você tenha realmente conseguido entender os conceitos abordados ao longo destas páginas. Programação é uma área fascinante, complexa, em constante evolução, e tenho certeza que as primeiras vezes que você teve contato com programação orientada a objetos deve ter ficado um pouco assustado(a) porque é um choque de realidade onde quando achávamos que dominávamos o essencial sobre uma determinada linguagem de programação, nos era mostrado que aquilo era apenas a ponta do iceberg.

Agora entendido o essencial de programação orientada a objetos em Python, hoje de tentar dar passos maiores e enfrentar desafios maiores, já que ainda há um mundo de possibilidades a serem descobertas.

Nunca se esqueça que programação é prática, prática, muito estudo e mais prática... e mesmo assim sempre terão situações que teremos de recorrer a documentação ou a fóruns especializados para buscar a resolução de algum problema. Importante salientar também que em Python, dependendo o propósito, existe uma infinidade de bibliotecas e módulos desenvolvidos a fim de automatizar ou facilitar certos processos, todas elas, com suas particularidades, também são estudadas as vezes do zero...

Sendo assim, apenas quero que fique bem claro que ao final deste livro, se você praticou em cima dos exemplos, se entendeu toda a lógica por traz de cada tópico abordado, você sim deu um grande passo em seu aprendizado de Python. Com esta bagagem de conhecimento é sim possível criar um universo programado, mas espero que este seja apenas mais um degrau alcançado em busca de objetivos maiores.

Como sempre digo ao final de meus livros, espero que a leitura deste tenha sido tão prazerosa quanto foi para mim escrevê-lo, e espero que tenha sido de grande valia para seu aprendizado.

Agradeço a compra deste material e lhe desejo muito sucesso em sua carreira profissional, e quem sabe um dia nos cruzamos por aí em algum bloco de código de algum projeto...

Um forte abraço.

Fernando Feltrin.

27 - Introdução / Livro 3 – Ciência de Dados e Aprendizado de Máquina

Nota do Autor

Seja muito bem vindo(a) ao mundo do Machine Learning, em tradução livre, Aprendizado de Máquina. Ao longo deste pequeno livro introdutório estaremos entendendo de forma simples e objetiva o que é essa área da computação, quais suas particularidades e aplicações. Assim como de forma procedural estaremos pondo em prática a codificação de modelos e estruturas de redes neurais artificiais, aprendizado de máquina e tudo o que está ao seu entorno.

Tenha em mente que este livro tem o intuito principal de desenvolver bases bastante sólidas sobre essa que em minha humilde opinião, é uma das mais incríveis áreas da computação. Independentemente se você já atua nessa área, ou se já é um programador, ou se é um entusiasta que por seus motivos está começando agora nessa área, fique tranquilo pois todo conteúdo será abordado de uma forma bastante simples, didática e prática, de modo a desmistificar muita informação errônea que existe nesse meio assim como de forma gradativa aprender de fato o que é ciência de dados por meio de suas aplicações.

Com dedicação e muita prática tenho certeza que ao final desse livro você terá plenas condições de, seja para fins de estudo ou profissionais, criar seus próprios modelos e/ou adaptar modelos já existentes contextualizando-os para seus devidos fins.

Um forte abraço.

Fernando Feltrin

Afinal de contas, o que é Ciência de Dados e Aprendizado de Máquina?

Na área de tecnologia temos muitas áreas de especialização, o que normalmente é desconhecido do público comum e até mesmo de alguns profissionais da área, pois dependendo da sua área de formação e atuação, costumam se ater a apenas o próprio nicho se especializando a fundo apenas nele. A grande variedade de subdivisões se dá não só pela grande variedade de cursos de nível técnico, superior e/ou especializações que existem nesse meio formando especialistas de cada área, mas principalmente porque cada segmento da tecnologia possui muitas áreas de especialização dentro de cada nicho e um universo de informação ainda a ser explorado.

Especificamente dentro da área da ciência da computação, nicho da programação, existem também muitas subdivisões onde encontramos desde a criação de programas e ferramentas que possuem propósito final de resolver algum problema de forma computacional até a análise e manipulação de dados de forma científica. Uma destas áreas específicas é a que trata das particularidades da chamada Inteligência Artificial, sub área esta que trata desde processos de automação até modelos que simulam tipos de inteligência para os mais diversos propósitos.

Na verdade, existem muitas áreas que podem englobar ou compartilhar conceitos umas com as outras, o que nos dias atuais torna difícil até mesmo tentar dividir as áreas da tecnologia ou categorizá-las individualmente de acordo com suas particularidades. Uma destas áreas da computação, independentemente de qual classe, chamada Data Science, ou em tradução livre, Ciência de Dados, está muito em alta nos dias atuais, porém sua idealização se deu em meados da década de 50 e ainda estamos longe de explorar todo seu real potencial.

Basicamente se idealizaram na década de 50 rotinas (algoritmos) e modelos computacionais para se trabalhar processando grandes volumes de dados fazendo o uso do poder computacional da época para seu devido processamento. Você já deve imaginar que naquela época não tínhamos poder computacional para pôr em prática tais conceitos, logo, muitas dessas ideias acabaram sendo engavetadas.

Porém décadas depois tivemos enormes saltos tecnológicos e finalmente hoje temos poder de processamento de sobra, fazendo o uso de GPUs de alta performance, clusters de processamento paralelo e métodos de processamento onde realmente conseguimos trabalhar com enormes volumes de dados para fins de se analisar e obter informações a partir dos mesmos, sem falar de outras tecnologias que foram criadas em paralelo que nos permitiram avançar de forma exponencial. Hoje já é realidade termos também automação para praticamente qualquer coisa, assim como hardware e software desenvolvidos para aumentar e melhorar processos até então realizados puramente por pessoas com base em sua baixa eficiência se comparado a um computador.

Sendo assim retomamos e aperfeiçoamos aqueles conceitos lá da longínqua década de 50 para hoje termos diferentes áreas de processamento de dados sendo cada vez mais usadas de forma a criar impacto no mundo real. Muitos autores inclusive veem essa área da computação como uma evolução natural da ciência estatística que, combinada a toda automação que desenvolvemos, resultou num salto tecnológico sem precedentes. A tecnologia está infiltrada em

tudo e todas as outras áreas, colaborando para que nestes dias atribulados consigamos criar e produzir de modo a suprir as demandas desse mundo moderno.

Como comentado anteriormente, o conceitual teórico de criar formas e métodos de fazer com que computadores “trabalhem por nós” já existe de muito tempo. Hoje conseguimos concretizar várias dessas ideias graças aos avanços tecnológicos que obtivemos, seja em estrutura física ou lógica, a ponto de literalmente estarmos criando literalmente inteligência artificial, sistemas embarcados, integrados na chamada internet das coisas, softwares que automatizam processos e não menos importante que os exemplos anteriores, a digitalização de muita informação analógica que ainda hoje trabalhamos, mesmo numa fase de grandes avanços tecnológicos, ainda vivemos essa transição.

Falando da aplicação de “dados”, raciocine que em outras épocas, ou pelo menos em outras eras tecnológicas, independente do mercado, se produzia com base na simples aplicação de modelos administrativos, investimento e com base em seu retorno, com um pouco de estatística se criavam previsões para o mercado a médio prazo. Nesse processo quase que de tentativa e erro tínhamos modelos de processos com baixa eficiência, alto custo e consequentemente desgaste por parte dos colaboradores.

Hoje independentemente de que área que estivermos abordando, existem ferramentas computacionais de automação que, sem grandes dificuldades conseguem elevar muito a eficiência dos processos num mundo cada vez mais acelerado e interconectado, sedento por produtividade, com enormes volumes de dados gerados e coletados a cada instante que podemos usar de forma a otimizar nossos processos e até mesmo prever o futuro dos mercados.

Para ficar mais claro, imagine o processo de desenvolvimento de uma vacina, desde sua lenta pesquisa até a aplicação da mesma, tudo sob enormes custos, muito trabalho de pesquisa, tentativa e erro para poucos e pífios resultados. Hoje, falando a nível de ciência, se evolui mais em um dia do que no passado em décadas, e cada vez mais a tecnologia irá contribuir para que nossos avanços em todas as áreas sejam de forma mais acelerada e precisa.

Seguindo com o raciocínio, apenas por curiosidade, um exemplo bastante interessante é o da criação da vacina contra o vírus da raiva. Idealizada e pesquisada por Louis Pasteur, posteriormente publicada e liberada em 1885, na época de sua pesquisa a história nos conta que em seu laboratório quando ele ia coletar amostras de tecidos, de animais em diferentes estágios da infecção para que fosse feita as devidas análises e testes. Para realização dos devidos procedimentos iam até o ambiente das cobaias ele, um assistente e era levado consigo uma espingarda... A espingarda era para matar ele caso houvesse sido contaminado, uma vez que naquela época esta ainda era uma doença fatal, de quadro clínico irreversível uma vez que a pessoa fosse contaminada.

Hoje conseguimos usar de poder computacional para reduzir tempo e custos (principalmente em pesquisa e no âmbito de criar modelos onde se usa de ferramentas computacionais para simular os cenários e as aplicações e seus respectivos resultados, em milhões de testes simulados), aplicar a vacina de forma mais adaptada às necessidades de uma população ou área geográfica e ainda temos condições de aprender com esses dados para melhorar ainda mais a eficiência de todo este processo para o futuro. Da mesma forma, existe muita pesquisa científica em praticamente todas as áreas de conhecimento usando das atuais ferramentas de

ciência de dados e das técnicas de aprendizagem de máquina para criar novos prospectos e obter exponenciais avanços na ciência.

Neste pequeno livro iremos abordar de forma bastante prática os principais conceitos desta gigante área de conhecimento, a fim de desmistificar e principalmente de elucidar o que realmente é esta área de inteligência artificial, ciência de dados e aprendizagem de máquina, além de suas vertentes, subdivisões e aplicações. Estaremos criando de forma simples e procedural uma bagagem de conhecimento sobre os principais tópicos sobre o assunto assim como estaremos pondo em prática linha a linha de código com suas devidas explicações para que ao final desse livro você tenha condições não só de trabalhar em cima de ferramentas de computação científica, mas criar seus próprios modelos de análise de dados e ter condições de os adaptar para suas aplicações diárias.

Como se trabalha com Ciência de Dados? Quais ferramentas utilizaremos?

Se tratando de ciência de dados, e como já mencionado anteriormente, estamos falando de um dos nichos da área de programação, logo, estaremos trabalhando em uma camada intermediária entre o usuário e o sistema operacional. Usando uma linguagem de programação e ferramentas criadas a partir dela para facilitar nossas vidas automatizando uma série de processos, que diga-se de passagem em outras épocas era necessário se criar do zero, teremos a faca e o queijo nas mãos para criar e pôr em uso nossos modelos de análise e processamento de dados.

Falando especificamente de linguagem de programação, estaremos usando uma das linguagens mais “queridinhas” dentro deste meio que é a linguagem Python. Python é uma linguagem de sintaxe simples e grande eficiência quando se trata de processamento em tempo real de grandes volumes de dados, dessa forma, cada tópico abordado e cada linha de código que iremos criar posteriormente será feito em Python. Usando dessa linguagem de programação e de uma série de ferramentas criadas a partir dela, como mencionei anteriormente, estaremos desmistificando muita coisa que é dita erroneamente sobre essa área, assim como mostrar na prática que programação voltada a ciência de dados não necessariamente é, ou deve ser, algo extremamente complexo, maçante de difícil entendimento e aplicabilidade.

Caso você seja mais familiarizado com outra linguagem de programação você não terá problemas ao adaptar seus algoritmos para Python, já que aqui estaremos trabalhando com a mesma lógica de programação e você verá que cada ferramenta que utilizaremos ainda terá uma sintaxe própria que indiretamente independe da linguagem abordada. Usaremos da linguagem Python, mas suas bibliotecas possuem ou podem possuir uma sub-sintaxe própria.

Especificamente falando das ferramentas que serão abordadas neste curso, estaremos usando de bibliotecas desenvolvidas e integradas ao Python como, por exemplo, a biblioteca Pandas, uma das melhores e mais usadas no que diz respeito a análise de dados (leitura, manipulação e visualização dos mesmos), outra ferramenta utilizada será o Numpy, também uma das melhores no que tange a computação científica e uso de dados por meio de operações matemáticas. Também faremos o uso da biblioteca SKLearn que dentro de si possui uma série de ferramentas para processamento de dados já numa sub área que trabalharemos que é a de aprendizagem de máquina. Outro exemplo é que estaremos usando muito da biblioteca Keras, que fornece ferramentas e funções para a fácil criação de estruturas de redes neurais. Por fim também estaremos fazendo o uso do TensorFlow, biblioteca esta desenvolvida para redes neurais artificiais com grande eficiência e facilidade de uso, além de uma forma mais abstrata (e lógica) de se analisar dados.

Importante apenas complementar que todas ferramentas usadas ao longo desse livro são de distribuição livre, baseadas em políticas opensource, ou seja, de uso, criação, modificação e distribuição livre e gratuita por parte dos usuários. Existem ferramentas proprietárias de grandes empresas mas aqui vamos nos ater ao que nos é fornecido de forma livre.

Quando e como utilizaremos tais ferramentas?

Se você está começando do zero nesta área fique tranquilo pois a sua enorme complexidade não necessariamente significará ter que escrever centenas de milhares de linhas de código, escritos em linguagem próxima a linguagem de máquina em algo extremamente complexo nem nada do tipo. Basicamente iremos definir que tipo de situação ou problema abordaremos de forma computacional e a partir deste, escolhemos o tipo de ferramenta mais eficiente para aquele fim. Em seguida, usando da linguagem Python dentro de uma IDE ou interpretador, criar e entender passo a passo, linha a linha de código de como estamos abstraindo um problema ou situação real para que, de forma computacional, possamos criar modelos para contextualizar, analisar, aprender com esses dados e criar previsões a partir dos mesmos..

Apenas para fins de exemplo, também, em ciência de dados basicamente trabalharemos com modelos lógicos onde faremos a leitura de dados (em certos casos, grandes volumes de dados), o polimento, a manipulação e a classificação dos mesmos. Também desenvolveremos ferramentas e até mesmo modelos de redes neurais artificiais para que a partir de etapas de treinamento e aprendizagem de máquina elas possam ler, interpretar dados, aprender com seus erros, chegar a resultados satisfatórios de aprendizado e fornecer previsões a partir destes, tudo de forma simples e objetiva.

Um dos exemplos práticos que estaremos abordando em detalhes é a criação e uso de uma rede neural que com base num banco de dados consegue identificar em exames de mamografia a probabilidade de uma determinada lesão identificada a partir do exame se tornar câncer de mama. Raciocine primeiramente que dentro desta área da Radiologia Médica (Diagnóstico por Imagem) existe o profissional técnico que realiza tal tipo de exame, posicionando as estruturas anatômicas a fim de obter imagens da anatomia por meio de radiações ionizantes, um médico especialista que interpreta a imagem com base em sua bagagem de conhecimento de anatomia e patologia e por fim equipamentos analógicos e digitais que produzem imagens da anatomia para fins diagnósticos.

Fazendo o uso de ferramentas de ciência de dados podemos criar um modelo de rede neural que irá aprender a reconhecer padrões de características patológicas a partir de um banco de dados de imagens de exames de raios-x. Dessa forma os profissionais desta área podem ter um viés de confirmação ou até mesmo chegar a taxas de precisão muito maiores em seu diagnóstico, agora possíveis de serem alcançadas por meio destas ferramentas. Vale lembrar que, neste exemplo, assim como em outros, o uso de poder computacional nunca substituirá o(s) profissional(is) em questão, a tecnologia na verdade sempre vem como um grande aliado para que se melhore este e outros processos, no final, todos saem ganhando.

Então, para resumir, se você já está habituado a trabalhar com ferramentas voltadas a programação ou se você está começando do zero, fique tranquilo pois basicamente estaremos usando a combinação de linguagem de programação e seu software interpretador para criar modelos de análise de dados, posteriormente poderemos trabalhar em cima desses modelos e/ou até mesmo distribuí-los para livre utilização de terceiros.

Qual a Abordagem que Será Utilizada?

Se tratando de uma área com tantas subdivisões, cada uma com suas grandes particularidades, poderíamos ir pelo caminho mais longo que seria entender de forma gradual o que é em teoria cada área, como uma se relaciona com outra, como umas são evoluções naturais de outras, etc... Porém, abordaremos o tema da forma mais simples, direta, didática e prática possível. Tenha em mente que devido a nossas limitações, neste caso, de um livro, não teremos como abordar com muita profundidade certos pontos, mas tenha em mente que tudo o que será necessário estará devidamente explicado no decorrer dessas páginas, assim como as referências necessárias para que se busque mais conhecimento caso haja necessidade.

Para se ter uma ideia, poderíamos criar extensos capítulos tentando conceituar programação em Python, inteligência artificial, ciência de dados e aprendizagem de máquina quanto aos tipos de problemas e respectivamente os tipos de soluções que conseguimos de forma computacional.

Poderíamos dividir o tema quanto às técnicas que usamos mais comumente (análise, classificação, regressão, previsão, visão computacional, etc...), ou pelo tipo de rede neural artificial mais eficiente para cada caso (de uma camada, múltiplas camadas, densas (deep learning), convolucionais, recorrentes, etc...), assim como poderíamos separar esse tema de acordo com as ferramentas que são mais usadas por entusiastas, estudantes e profissionais da área (SciKitLearn, Pandas, Numpy, TensorFlow, Keras, OpenCV, etc...).

Ainda poderíamos também fazer uma abordagem de acordo com os métodos aplicados sobre dados (classificação binária, classificação multiclasse, regressão linear, regressão logística, balanço viés-variança, KNN, árvores de decisão, SVMs, clusterização, sistemas de recomendação, processamento de linguagem natural, deep learning, etc...) mas ainda assim seria complicado ao final de todo know hall teórico conseguir fazer as interconexões entre todos conceitos de forma clara.

Sendo assim, a maneira como iremos abordar cada tópico de cada tema de cada nicho será totalmente prático. Usaremos de exemplos reais, de aplicação real, para construir passo a passo, linha a linha de código o conhecimento necessário para se entender de fato o que é ciência de dados e os métodos de treinamento, aprendizagem de máquina para podermos criar e adaptar nossas próprias ferramentas para estes exemplos e para os próximos desafios que possam surgir.

Ao final deste breve livro sobre o assunto você terá bagagem de conhecimento suficiente para criar suas próprias ferramentas de análise de dados e de fato, a partir desses, transformar dados em informação.

28 – Preparação do Ambiente de Trabalho

Uma vez que estaremos trabalhando com programação, independente do sistema operacional que você usa em sua rotina, será necessário realizar as instalações de algumas ferramentas que nativamente não fazem parte do seu sistema operacional para que de fato possamos iniciar com a programação. Inicialmente o que você fará é a instalação do Python 3 em sua última versão, posteriormente iremos instalar algumas bibliotecas e módulos complementares ao Python para que possamos fazer uso dessas ferramentas pré-programadas e configuradas, ao invés de criar tudo do zero.

Instalação do Python 3

A linguagem Python, assim como suas ferramentas, nativamente não fazem parte dos pacotes pré instalados no Windows, porém é bastante simples realizar a sua instalação para que possamos ter acesso as suas ferramentas de codificação.

Primeiramente abrindo o seu navegador e digitando Python 3 você encontrará facilmente o site do projeto que mantém essa ferramenta open source.

A screenshot of a Google search results page. The search query 'python 3' is entered in the search bar. Below the search bar, there are tabs for 'Todas', 'Imagens', 'Notícias', 'Vídeos', 'Livros', 'Mais', 'Configurações', and 'Ferramentas'. The 'Todas' tab is selected. The search results show approximately 395,000,000 results found in 0.48 seconds. The top result is a link to 'Python 3.0 Release | Python.org' with the URL <https://www.python.org/download/releases/3.0/>. The snippet of the page content indicates that Python 3.0 is end-of-life and users should upgrade to Python 3.1.

Entrando no site www.python.org você terá acesso a página inicial onde constam as informações das atualizações além dos links mais relevantes.

A screenshot of the Python.org homepage. The header features a navigation bar with links for 'Python', 'PSF', 'Docs', 'PyPI', 'Jobs', and 'Community'. Below the header is a search bar with a 'Donate' button and a 'GO' button. The main content area has a sidebar with 'Tweets by @ThePSF' and a message from the PSF about grants to events. The main content area features a section titled 'Python 3.0 Release' with a sub-section for 'Python 3.0'. It contains text stating that Python 3.0 is end-of-life and users should upgrade to Python 3.1, along with a link to the download pages. There is also a note that Python 3.0 has been replaced by Python 3.1.

Repare que no menu superior temos uma guia Downloads, passando o mouse em cima da mesma é aberto um menu com a indicação da versão mais apropriada para download.

The screenshot shows the Python.org homepage with a focus on the 'Downloads' section for Windows. A large yellow button labeled 'Download Python 3.7.3' is prominently displayed. Above this button, a note states: 'Note that Python 3.5+ cannot be used on Windows XP or earlier.' Below the main download button, there's a link to 'View the full list of downloads.'

No meu caso, rodando Windows 10 ele identifica e recomenda que a versão mais adequada para meu sistema operacional é a Python 3.7.3, clicando no botão o download é iniciado. Caso essa identificação não esteja correta você pode clicar no link View the full list of downloads, nele será apresentado a lista com as diferentes versões para os respectivos sistemas operacionais.

The screenshot shows the Python.org homepage with a focus on the 'Downloads' section for Windows. A large yellow button labeled 'Download Python 3.7.3' is prominently displayed. Above this button, a note states: 'Note that Python 3.5+ cannot be used on Windows XP or earlier.' Below the main download button, there's a link to 'View the full list of downloads.'

Note que abaixo do botão Download Python 3.7.3 existem links que redirecionam para página específica para Linux/Unix, Mac OS e outros sistemas operacionais.

O arquivo de instalação tem por volta de 25Mb e conta com um instalador próprio que fará todo processo automaticamente assim como a maioria dos instaladores. Apenas teremos de ter cuidado a algumas configurações simples.



Nesta tela inicial do instalador temos que ficar atentos ao seguinte: Clicando em Install Now o instalador fará a instalação padrão com os diretórios convencionais. Caso você queira usar um diretório diferente, ou caso você queira instalar separadamente diferentes versões do Python você deve escolher Customize installation e definir um diretório manualmente. Por fim, marcando a opção Add Python 3.7 to PATH você estará criando os registros necessários para que o Windows considere o Python a linguagem de programação padrão para seus interpretadores. (Esta opção pode ser definida ou configurada manualmente posteriormente.).

Terminada a instalação talvez seja necessário reiniciar o sistema para que ele de fato assimile o interpretador ao registro. Após isso você já pode usar normalmente qualquer IDE para ler e criar seus códigos em Python.

Instalação da Suite Anaconda

Assim como a instalação do Python, usaremos um método parecido para instalar em nosso sistema a suíte de aplicativos Anaconda. Quem trabalha com computação científica e Python normalmente está familiarizado com essa suíte que conta com uma série de ferramentas instaladas e pré-configuradas de forma a facilitar seu uso.

Outra vantagem de se usar o Anaconda é a sua integração interna e externa, enquanto usar ferramentas separadas podem gerar certos conflitos entre si, as ferramentas disponíveis no Anaconda são perfeitamente integradas e funcionais entre si.

No decorrer deste curso estaremos usando muito, por exemplo, o Spyder, uma IDE que permite a execução de códigos assim como sua visualização em tempo real.

Partindo para o que interessa, abrindo seu navegador e digitando python anaconda você encontrará facilmente a página oficial deste projeto.

Aproximadamente 17.200.000 resultados (0,65 segundos)

Anaconda Python/R Distribution - Anaconda
https://www.anaconda.com/distribution/ ▾ Traduzir esta página
The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over ...
Anaconda Distribution · Why Anaconda · Anaconda · Anaconda Enterprise · Blog
Você visitou esta página 2 vezes. Última visita: 14/05/19

Abrindo a página do projeto, assim como anteriormente na página do Python, ela identificará seu sistema operacional e com base nisso irá sugerir o download mais apropriado

The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 11 million users worldwide, it is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:

- Quickly download 1,500+ Python/R data science packages
- Manage libraries, dependencies, and environments with Conda
- Develop and train machine learning and deep learning models with scikit-learn, TensorFlow, and Theano
- Analyze data with scalability and performance with Dask, NumPy, pandas, and Numba

The open-source Anaconda Distribution is the easiest way to perform Python/R data science and machine learning on Linux, Windows, and Mac OS X. With over 11 million users worldwide, it is the industry standard for developing, testing, and training on a single machine, enabling *individual data scientists* to:

- Quickly download 1,500+ Python/R data science packages
- Manage libraries, dependencies, and environments with Conda
- Develop and train machine learning and deep learning models with scikit-learn, TensorFlow, and Theano
- Analyze data with scalability and performance with Dask, NumPy, pandas, and Numba

Clicando no botão Download a página irá rolar para a seção adequada ao seu sistema operacional.



Anaconda 2019.03 for Windows Installer

Python 3.7 version

[Download](#)

64-Bit Graphical Installer (662 MB)
32-Bit Graphical Installer (546 MB)

Python 2.7 version

[Download](#)

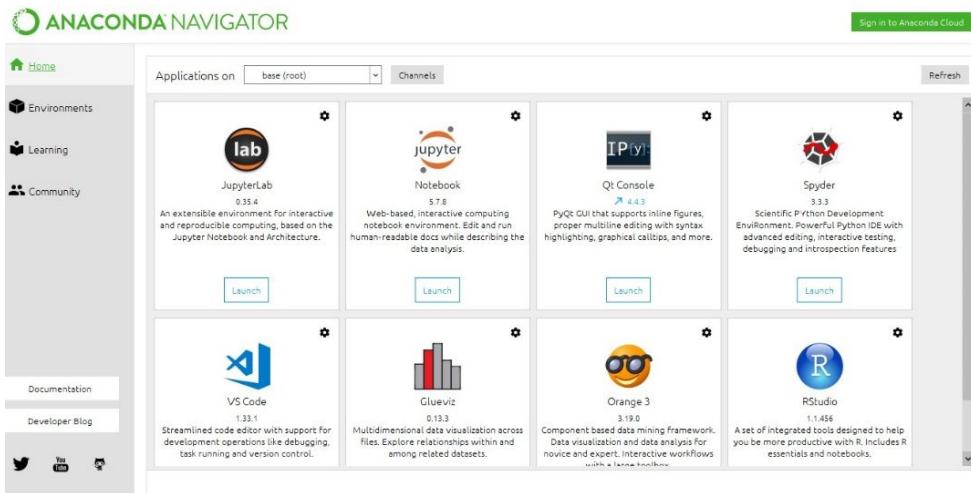
64-Bit Graphical Installer (587 MB)
32-Bit Graphical Installer (493 MB)

Após fazer o download de acordo com a versão do Python (3.7), o seu sistema operacional e tipo de arquitetura, simplesmente faremos a instalação por meio de seu instalador padrão.



A instalação é basicamente avançar as etapas e, assim como antes, alterar o diretório de instalação caso seja necessário. Uma vez terminada a instalação você terá acesso a suíte pelo seu atalho criado no menu iniciar.

Abrindo o Anaconda Navigator você tem acesso as principais ferramentas (elas podem ser abertas separadamente via menu iniciar sem problema).



Apenas entendendo o que são essas ferramentas, ao longo do curso usaremos o Spyder que como mencionado anteriormente, é uma IDE de fácil usabilidade que permite a execução de códigos assim como realizar a análise dos mesmos em tempo real. Temos a disposição também o VSCode, uma IDE leve que conta com uma grande variedade de plugins que adicionam maiores funcionalidades a si. Por fim temos o Jupyter Notebook, muito usado pela facilidade de ser uma IDE que roda a partir do navegador, além de permitir trabalhar com blocos de códigos divididos em células independentes.

A suíte Anaconda ainda conta com outras ferramentas, porém neste curso, para os devidos fins, estaremos usando apenas o Spyder, podendo também fazer alguns procedimentos via VSCode ou Jupyter Notebook.

Instalação das Bibliotecas e Módulos Adicionais

Python 3 e Anaconda instalados, vamos apenas fazer o download e a importação de algumas bibliotecas que usaremos ao longo desse curso e que nativamente não estão integradas a instalação padrão do Python.

Apenas para fins de exemplo, raciocine que posteriormente quando adentramos a fase de codificação, estaremos usando a linguagem Python, dentro da IDE Spyder, fazendo uso de ferramentas de machine learning como:

Numpy - Biblioteca para computação científica que oferece uma série de ferramentas para tratamento, aplicação de fórmulas aritméticas e uso de dados em forma vetorial e matricial.

Pandas - Biblioteca que oferece ferramentas para tratamento de grandes volumes de dados com alta performance de processamento.

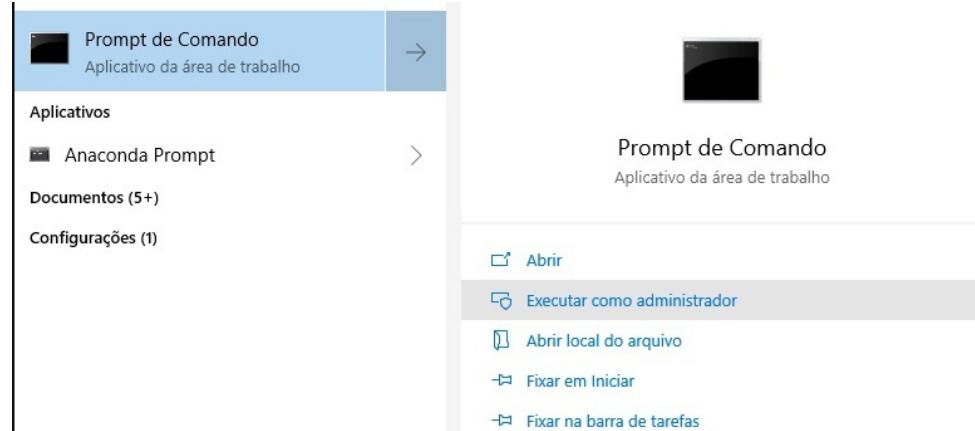
SciKitLearn - Biblioteca desenvolvida a partir do Numpy que oferece uma série de ferramentas para mineração e análise de dados.

Keras - Biblioteca que permite a fácil criação e configuração de estruturas de redes neurais artificiais.

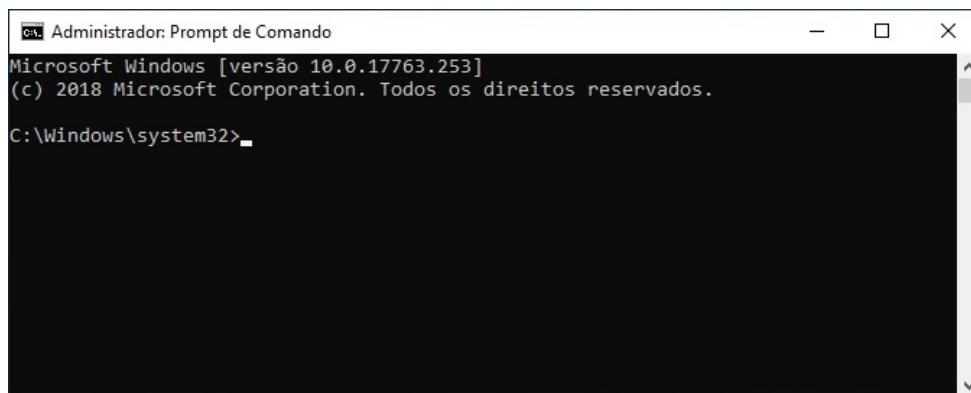
TensorFlow - Biblioteca que permite a abstração de algoritmos de machine learning para que os mesmos sejam criados em modelo de fluxo de processamento, aumentando assim a performance quando trabalhado com grandes volumes de dados.

Graças ao Anaconda, em seu sistema foi instalado uma ferramenta chamada comumente de pip que facilita a instalação e integração dessas bibliotecas por meio de simples comandos via terminal.

Vá até o menu Iniciar e digite CMD, execute o mesmo como administrador.



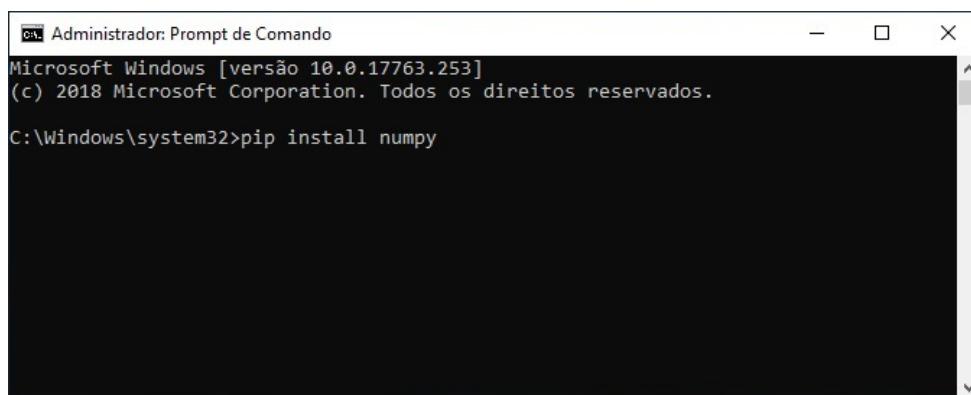
Uma vez aberto o Prompt de Comando, iremos fazer as devidas importações, instalações e atualizações via código.



```
Administrador: Prompt de Comando
Microsoft Windows [versão 10.0.17763.253]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\Windows\system32>
```

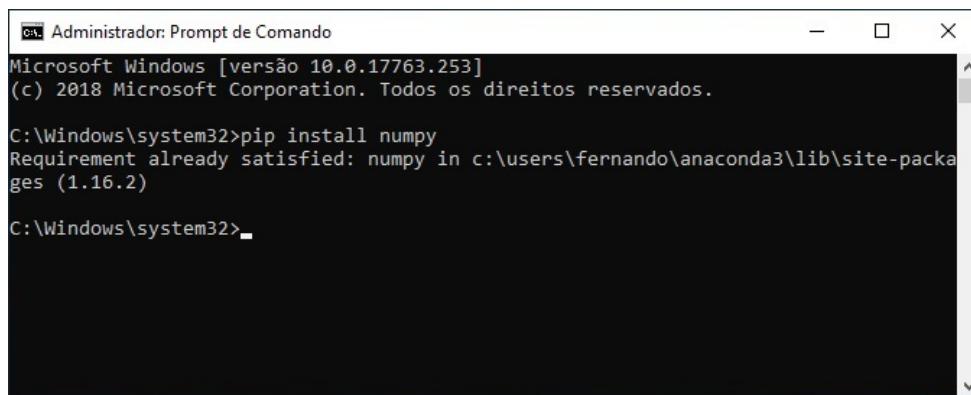
Inicialmente, para realizar a importação e integração da biblioteca Numpy basta executar o comando pip install numpy.



```
Administrador: Prompt de Comando
Microsoft Windows [versão 10.0.17763.253]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\Windows\system32>pip install numpy
```

Após a instalação você não terá nenhum retorno no terminal, mas uma vez feita a instalação essa biblioteca já pode ser instanciada dentro de um código via IDE.



```
Administrador: Prompt de Comando
Microsoft Windows [versão 10.0.17763.253]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\Windows\system32>pip install numpy
Requirement already satisfied: numpy in c:\users\fernando\anaconda3\lib\site-packages (1.16.2)

C:\Windows\system32>
```

Para a instalação das outras bibliotecas basta repetir o processo, agora executando via CMD as linhas de comando abaixo:

```
pip install pandas
pip install scikit-learn
pip install keras
pip install seaborn
```

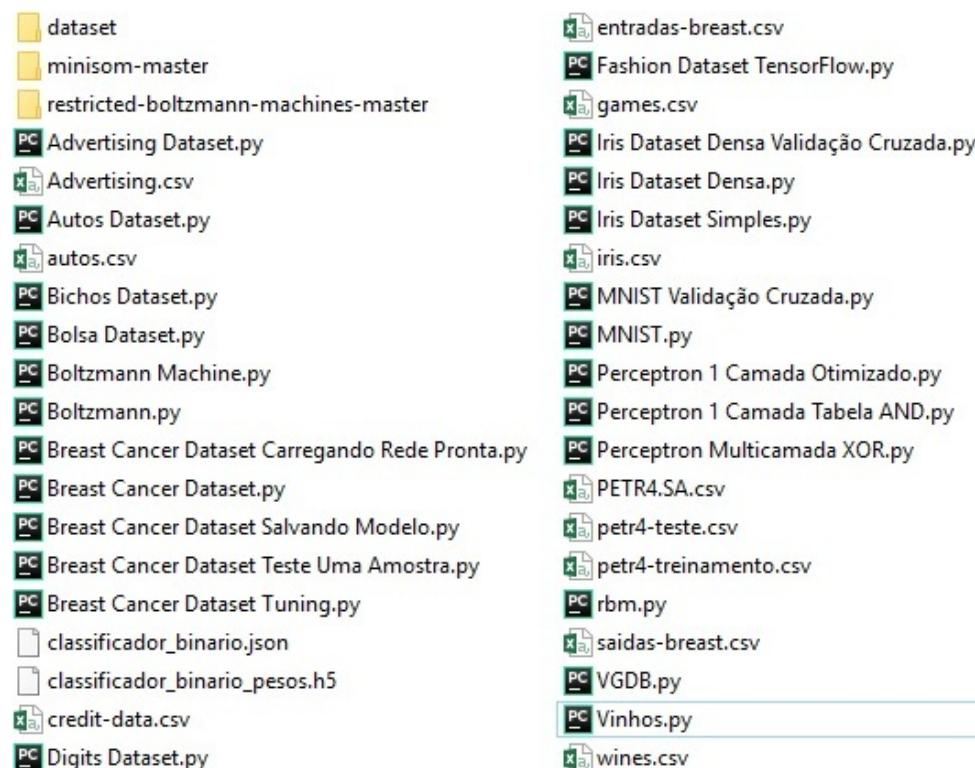
```
pip install tensorflow  
pip install minisom
```

Caso você tenha algum problema ao realizar essas importações via CMD uma alternativa é usar o terminal instalador do Anaconda, substituindo pip por conda via CMD você estará instanciando o terminal do Anaconda. Ex: conda install scikit-learn.

Se mesmo assim ocorreram erros no processo de importação, todas essas bibliotecas em seu site possuem seu devido instalador de acordo com o sistema operacional, apenas ficar atento ao instalar manualmente a definição correta dos diretórios.

Todos os demais códigos e ferramentas podem ser baixados diretamente em:

[Bit.ly/2OEUB0T](https://bit.ly/2OEUB0T)



Ou por meio de meu repositório no GitHub:

<https://github.com/fernandofeltrin/Ciencia-de-Dados-e-Aprendizado-de-Maquina>



fernandofelrin Add files via upload

- [Advertising Dataset \(Previsão Quantitativa\).py](#)
- [Autos Dataset \(Regressão de Dados de Planilhas Excel\).py](#)
- [Bichos Dataset \(Classificação de Imagens de Animais\).py](#)
- [Bolsa Dataset \(Previsão de Séries Temporais\).py](#)
- [Boltzmann Machine \(Sistemas de Recomendação\).py](#)
- [Breast Cancer Dataset \(Classificação Binária Via Rede Neural\).py](#)
- [Breast Cancer Dataset Carregando Rede Pronta.py](#)
- [Breast Cancer Dataset Salvando Modelo.py](#)
- [Breast Cancer Dataset Teste Uma Amostra.py](#)
- [Breast Cancer Dataset Tuning.py](#)
- [Digits Dataset \(Reconhecimento de Caracteres\).py](#)
- [Fashion Dataset TensorFlow \(Classificação a Partir de Imagens\).py](#)
- [Iris Dataset Densa \(Classificação Multiclasse Via Rede Neural Densa\).py](#)
- [Iris Dataset Densa Validação Cruzada.py](#)
- [Iris Dataset Simples \(Classificação Multiclasse Via Rede Neural Simples\).py](#)
- [MNIST Validação Cruzada.py](#)
- [MNIST.py](#)
- [Perceptron 1 Camada Otimizado.py](#)
- [Perceptron 1 Camada Tabela AND.py](#)
- [Perceptron Multicamada XOR.py](#)
- [VGDB \(Regressão com Múltiplas Saídas\).py](#)
- [Vinhos \(Mapas Auto Organizáveis\).py](#)

29 – Teoria e Prática em Ciência de Dados

Teoria Sobre Redes Neurais Artificiais

Um dos métodos de processamento de dados que iremos usar no decorrer de todos exemplos deste livro, uma vez que sua abordagem será prática, é o de redes neurais artificiais. Como o nome já sugere, estamos criando um modelo computacional para abstrair uma estrutura anatômica real, mais especificamente um ou mais neurônios, células de nosso sistema nervoso central e suas interconexões.

Nosso sistema nervoso, a nível celular, é composto por neurônios e suas conexões chamadas sinapses. Basicamente um neurônio é uma estrutura que a partir de reações bioquímicas que o estimular geram um potencial elétrico para ser transmitido para suas conexões, de forma a criar padrões de conexão entre os mesmos para ativar algum processo de outro subsistema ou tecido ou simplesmente guardar padrões de sinapses no que convencionalmente entendemos como nossa memória.

Na eletrônica usamos de um princípio parecido quando temos componentes eletrônicos que se comunicam em circuitos por meio de cargas de pulso elétrico gerado, convertido e propagado de forma controlada e com diferentes intensidades a fim de desencadear algum processo ou ação.

Na computação digital propriamente dita temos modelos que a nível de linguagem de máquina (ou próxima a ela) realizam chaveamento traduzindo informações de código binário para pulso ou ausência de pulso elétrico sobre portas lógicas para ativação das mesmas dentro de um circuito, independentemente de sua função e robustez.

Sendo assim, não diferentemente de outras áreas, aqui criaremos modelos computacionais apenas como uma forma de abstrair e simplificar a codificação dos mesmos, uma vez que estaremos criando estruturas puramente lógicas onde a partir delas teremos condições de interpretar dados de entrada e saídas de ativação, além de realizar aprendizagem de máquina por meio de reconhecimento de padrões e até mesmo desenvolver mecanismos de interação homem-máquina ou de automação.

A nível de programação, um dos primeiros conceitos que precisamos entender é o de um Perceptron. Nome este para representar um neurônio artificial, estrutura que trabalhará como modelo lógico para interpretar dados matemáticos de entrada, pesos aplicados sobre os mesmos e funções de ativação para ao final do processo, assim como em um neurônio real, podermos saber se o mesmo é ativado ou não em meio a um processo.

Aproveitando o tópico, vamos introduzir alguns conceitos que serão trabalhados posteriormente de forma prática, mas que por hora não nos aprofundaremos, apenas começaremos a construir aos poucos uma bagagem de conhecimento sobre tais pontos.

Quando estamos falando em redes neurais artificiais, temos de ter de forma clara que este conceito é usado quando estamos falando de toda uma classe de algoritmos usados para encontrar e processar padrões complexos a partir de bases de dados usando estruturas lógicas chamadas perceptrons. Uma rede neural possui uma estrutura básica de entradas, camadas de processamento, funções de ativação e saídas. De forma geral é composta de neurônios artificiais,

estruturas que podem ser alimentadas pelo usuário ou retroalimentadas dentro da rede por dados de suas conexões com outros neurônios, aplicar alguma função sobre esses dados e por fim gerar saídas que por sua vez podem representar uma tomada de decisão, uma classificação, uma ativação ou desativação, etc...

Outro ponto fundamental é entender que assim como eu uma rede neural biológica, em uma rede neural artificial haverão meios de comunicação entre esses neurônios e podemos inclusive usar esses processos, chamados sinapses, para atribuir valores e funções sobre as mesmas. Uma sinapse normalmente possui uma sub estrutura lógica chamada Bias, onde podem ser realizadas alterações intermediárias aos neurônios num processo que posteriormente será parte essencial do processo de aprendizado de máquina já que é nessa “camada” que ocorrem atualizações de valores para aprendizado da rede.

Em meio a neurônios artificiais e suas conexões estaremos os separando virtualmente por camadas de processamento, dependendo da aplicação da rede essa pode ter apenas uma camada, mais de uma camada ou até mesmo camadas sobre camadas (deep learning). Em suma um modelo básico de rede neural conterá uma camada de entrada, que pode ser alimentada manualmente pelo usuário ou a partir de uma base de dados. Em alguns modelos haverão o que são chamadas camadas ocultas, que são intermediárias, aplicando funções de ativação ou funcionando como uma espécie de filtros sobre os dados processados. Por fim desde os moldes mais básicos até os mais avançados sempre haverá uma ou mais camadas de saída, que representam o fim do processamento dos dados dentro da rede após aplicação de todas suas funções.

A interação entre camadas de neurônios artificiais normalmente é definida na literatura como o uso de pesos e função soma. Raciocine que a estrutura mais básica terá sempre esta característica, um neurônio de entrada tem um valor atribuído a si mesmo, em sua sinapse, em sua conexão com o neurônio subsequente haverá um peso de valor gerado aleatoriamente e aplicado sobre o valor inicial desse neurônio em forma de multiplicação. O resultado dessa multiplicação, do valor inicial do neurônio pelo seu peso é o valor que irá definir o valor do próximo neurônio, ou em outras palavras, do(s) que estiver(em) na próxima camada. Seguindo esse raciocínio lógico, quando houverem múltiplos neurônios por camada além dessa fase mencionada anteriormente haverá a soma dessas multiplicações.

$$1 \text{ neurônio} \quad Z = \text{entradas} \times \text{pesos}$$

$$n \text{ neurônios} \quad Z = (\text{entrada1} \times \text{peso1}) + (\text{entrada2} \times \text{peso2}) + \text{etc...}$$

Por fim um dos conceitos introdutórios que é interessante abordarmos agora é o de que mesmo nos modelos mais básicos de redes neurais haverão funções de ativação. Ao final do processamento das camadas é executada uma última função, chamada de função de ativação, onde o valor final poderá definir uma tomada de decisão, uma ativação (ou desativação) ou classificação. Esta etapa pode estar ligada a decisão de problemas lineares (0 ou 1, True ou False, Ligado ou Desligado) ou não lineares (probabilidade de 0 ou probabilidade de 1). Posteriormente ainda trabalharemos com conceitos mais complexos de funções de ativação.

Linear - ReLU (Rectified Linear Units) - Onde se X for maior que 0 é feita a ativação do neurônio, caso contrário, não. Também é bastante comum se usar a Step Function (Função

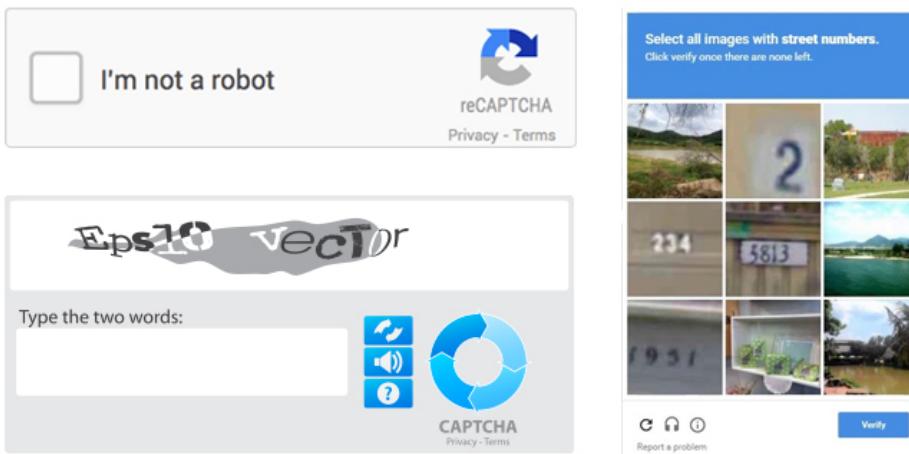
Degrau) onde é definido um valor como parâmetro e se o valor de X for igual ou maior a esse valor é feita a ativação do neurônio, caso contrário, não.

Non Linear - Sigmoid - Probabilidade de ativação ou de classificação de acordo com a proximidade do valor de X a 0 ou a 1. Apresenta e considera os valores intermediários entre 0 e 1 de forma probabilística.

Também estaremos trabalhando com outras formas de processamento quando estivermos realizando o treinamento supervisionado de nossa rede neural artificial. Raciocine que os tópicos apresentados anteriormente apenas se trata da estrutura lógica ao qual sempre iremos trabalhar, o processo de aprendizado de máquina envolverá mais etapas onde realizaremos diferentes processos em cima de nossa rede para que ela “aprenda” os padrões os quais queremos encontrar em nossos dados e gerar saídas a partir dos mesmos.

Aprendizado de Máquina

Quando estamos falando em Ciência e Dados, mais especificamente de Aprendizado de Máquina, é interessante começarmos a de fato entender o que essas nomenclaturas significam, para desmistificarmos muita informação errônea que existe a respeito desse termo. Na computação, apesar do nome “máquina”, basicamente estaremos criando modelos lógicos onde se é possível simular uma inteligência computacional, de forma que nosso software possa ser alimentado com dados de entrada e a partir destes, reconhecer padrões, realizar diversos tipos de tarefas sobre estes, aprender padrões de experiência e extrair informações ou desencadear tomadas de decisões a partir dos mesmos. Sendo assim, ao final temos um algoritmo funcional que foi treinado e aprendeu a atingir um objetivo com base nos dados ao qual foi alimentado.



Você pode não ter percebido, mas uma das primeiras mudanças na web quando começamos a ter sites dinâmicos foi o aparecimento de ferramentas de validação como os famosos captcha. Estas aplicações por sua vez tem uma ideia genial por trás, a de que pedindo ao usuário que realize simples testes para provar que ele “não é um robô”, eles estão literalmente treinando um robô a reconhecer padrões, criando dessa forma uma espécie de inteligência artificial. Toda vez que você responde um captcha, identifica ou sinaliza um objeto ou uma pessoa em uma imagem, você está colaborando para o aprendizado de uma máquina.

Quando criamos estes tipos de modelos de algoritmos para análise e interpretação de dados (independe do tipo), inicialmente os fazemos de forma manual, codificando um modelo a ser usado e reutilizado posteriormente para problemas computacionais de mesmo tipo (identificação, classificação, regressão, previsão, etc...).

Neste processo basicamente podemos ter três tipos diferentes e independentes de aprendizado de máquina:

Aprendizado Supervisionado - Método onde basicamente teremos dados de entrada e de saída e o que fazemos é manualmente treinar uma rede neural artificial para reconhecer e aprender os padrões que levaram até aquela saída. (Método usado indiretamente em captcha).

Aprendizado Não Supervisionado - Método onde com base em dados acumulados com diversas informações de cada amostra podemos fazer o seu processamento via rede neural para que a rede reconheça e agrupe essas informações quanto ao seu tipo ou classificação ou um determinado padrão de forma natural.

Aprendizado Por Reforço - Método onde uma rede processa padrões de dados para que a partir dos mesmos tente aprender qual a melhor decisão a ser tomada em meio a um processo ou uma classificação. Normalmente aprende quando identifica erros, assim, reserva os padrões de acertos e dá mais peso a eles no processamento.

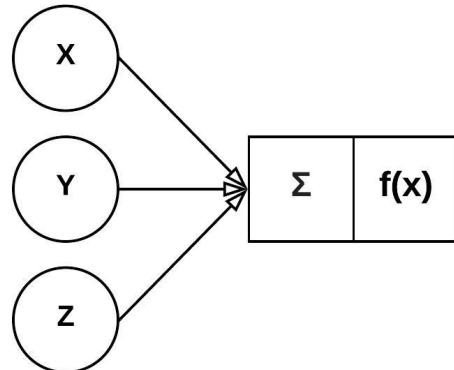
Existem outros modelos que não são comumente usados ou que são muito particulares para alguma situação, mas basicamente com o entendimento dos tipos de aprendizado citado acima podemos começar a montar a estrutura lógica ao qual usaremos para analisar e processar nossos dados.

Perceptrons

Para finalmente darmos início a codificação de tais conceitos, começaremos com a devida codificação de um perceptron de uma camada, estrutura mais básica que teremos e que ao mesmo tempo será nosso ponto inicial sempre que começarmos a programar uma rede neural artificial para problemas de menor complexidade, chamados linearmente separáveis.

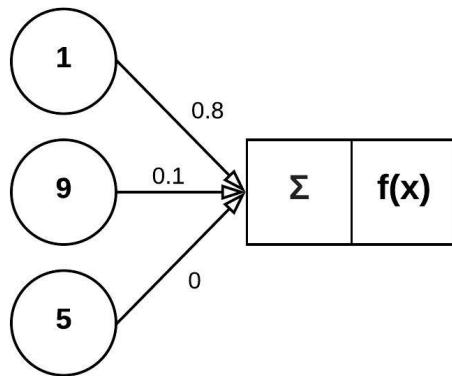
O modelo mais básico usado na literatura para fins didáticos é o modelo abaixo, onde temos a representação de 3 espaços alocados para entradas (X , Y e Z), note que estas 3 figuras se conectam com uma estrutura central com o símbolo Sigma Σ , normalmente atribuído a uma Função Soma, operação aritmética bastante comum e por fim, esta se comunica com uma última estrutura lógica onde há o símbolo de uma Função de Ativação $f(x)$.

Em suma, estaremos sempre trabalhando com no mínimo essas três estruturas lógicas, haverão modelos muito mais complexos ou realmente diferentes em sua estrutura, mas por hora o que deve ficar bem entendido é que todo perceptron terá entradas, operações realizadas sobre estas e uma função que resultará na ativação ou não deste neurônio em seu contexto.



A partir deste modelo podemos resolver de forma computacional os chamados problemas linearmente separáveis. De forma bastante geral podemos entender tal conceito como o tipo de problema computacional onde se resulta apenas um valor a ser usado para uma tomada de decisão. Imagine que a partir desse modelo podemos criar uma pequena rede neural que pode processar dados para que se ative ou não um neurônio, podemos abstrair essa condição também como fazímos com operadores lógicos, onde uma tomada de decisão resultava em 0 ou 1, True ou False, return x ou return y, etc... tomadas de decisão onde o que importa é uma opção ou outra.

Partindo para prática, agora atribuindo valores e funções para essa estrutura, podemos finalmente começar a entender de forma objetiva o processamento da mesma:



Sendo assim, usando o modelo acima, aplicamos valores aos nós da camada de entrada, pesos atribuídos a elas, uma função de soma e uma função de ativação. Aqui, por hora, estamos atribuindo manualmente esses valores de pesos apenas para exemplo, em uma aplicação real eles podem ser iniciados zerados, com valores aleatórios gerados automaticamente ou com valores temporários a serem modificados no processo de aprendizagem.

A maneira que faremos a interpretação inicial desse perceptron, passo-a-passo, é da seguinte forma:

Temos três entradas com valores 1, 9 e 5, e seus respectivos pesos 0.8, 0.1 e 0.

Inicialmente podemos considerar que o primeiro neurônio tem um valor baixo mas um impacto relativo devido ao seu peso, da mesma forma o segundo neurônio de entrada possui um valor alto, 9, porém de acordo com seu peso ele gera menor impacto sobre a função, por fim o terceiro neurônio de entrada possuir valor 5, porém devido ao seu peso ser 0 significa que ele não causará nenhum efeito sobre a função.

Em seguida teremos de executar uma simples função de soma entre cada entrada e seu respectivo peso e posteriormente a soma dos valores obtidos a partir deles.

Neste exemplo essa função se dará da seguinte forma:

$$\text{Soma} = (1 * 0.8) + (9 * 0.1) + (5 * 0)$$

$$\text{Soma} = 0.8 + 0.9 + 0$$

$$\text{Soma} = 1.7$$

Por fim a função de ativação neste caso, chamada de Step Function (em tradução livre Função Degrau) possui uma regra bastante simples, se o valor resultante da função de soma for 1 ou maior que 1, o neurônio será ativado, se for um valor abaixo de 1 (mesmo aproximado, mas menor que 1) este neurônio em questão não será ativado.

Step Function = Soma $\Rightarrow 1$ (Ativação)

Step Function = Soma < 1 (Não Ativação)

Note que esta é uma fase manual, onde inicialmente você terá de realmente fazer a alimentação dos valores de entrada e o cálculo destes valores manualmente, posteriormente entraremos em exemplos onde para treinar uma rede neural teremos inclusive que manualmente corrigir erros quando houverem para que a rede possa “aprender” com essas modificações.

Outro ponto importante é que aqui estamos apenas trabalhando com funções básicas que resultarão na ativação ou não de um neurônio, haverão situações que teremos múltiplas entradas, múltiplas camadas de funções e de ativação e até mesmo múltiplas saídas...

Última observação que vale citar aqui é que o valor de nossa Step Function também pode ser alterado manualmente de acordo com a necessidade.

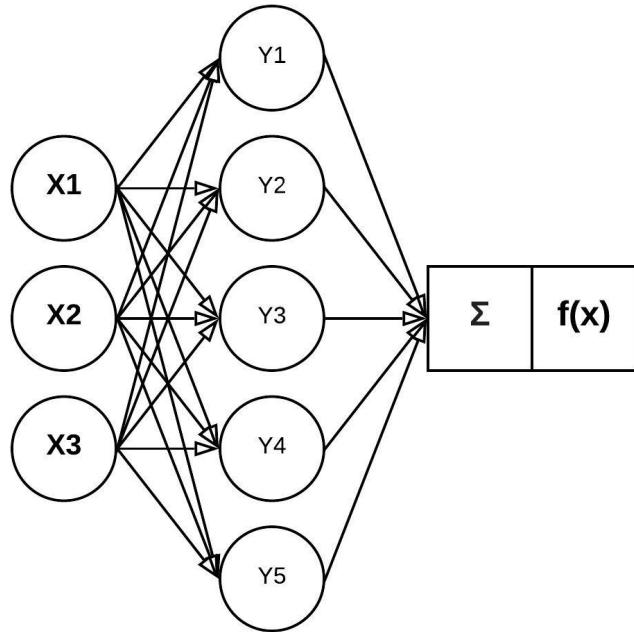
Com base nesse processamento podemos concluir que de acordo com o parâmetro setado em nossa Step Function, este perceptron em sua aplicação seria ativado, desencadeando uma tomada de decisão ou classificação de acordo com o seu propósito.

Perceptron Multicamada

Uma vez entendida a lógica de um perceptron, e exemplificado em cima de um modelo de uma camada, hora de aumentarmos um pouco a complexidade, partindo para o conceito de perceptron multicamada.

O nome perceptron multicamada por si só é bastante sugestivo, porém é importante que fique bem claro o que seriam essa(s) camada(s) adicionais. Como entendido anteriormente, problemas de menor complexidade que podem ser abstraídos sobre o modelo de um perceptron tendem a seguir sempre a mesma lógica, processar entradas e seus respectivos pesos, executar uma função (em perceptron de uma camada normalmente uma função de soma) e por fim a aplicação de uma função de ativação (em perceptron de uma camada normalmente uma função degrau, multicamada podemos fazer uso de outras como função sigmoide por exemplo).

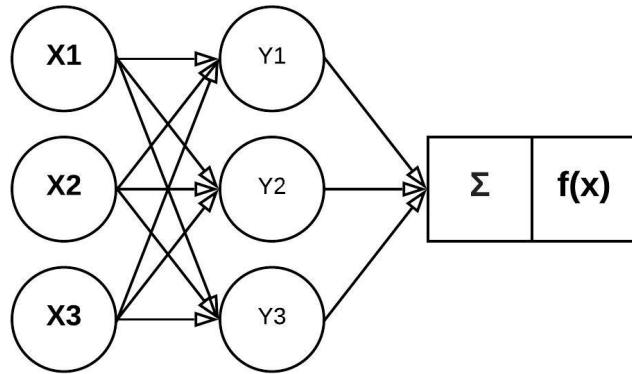
Agora, podemos adicionar o que por convenção é chamado de camada oculta em nosso perceptron, de forma que o processamento dos dados de entrada com seus pesos, passarão por uma segunda camada para ajuste fino dos mesmos, para posteriormente passar por função e ativação.



Repare no modelo acima, temos entradas representadas por X1, X2 e X3 e suas conexões com todos neurônios de uma segunda camada, chamada camada oculta, que se conecta com o neurônio onde é aplicada a função de soma e por fim com a fase final, a fase de ativação.

Além desse diferencial na sua representatividade, é importante ter em mente que assim como o perceptron de uma camada, cada neurônio de entrada terá seus respectivos pesos, e posteriormente, cada nó de cada neurônio da camada oculta também terá um peso associado.

Dessa forma, a camada oculta funciona basicamente como um filtro onde serão feitos mais processamentos para o ajuste dos pesos dentro desta rede.



Apenas para fins de exemplo, note que a entrada X_1 tem 3 pesos associados que se conectam aos neurônios Y_1 , Y_2 e Y_3 , que por sua vez possuem pesos associados à sua conexão com a função final a ser aplicada.

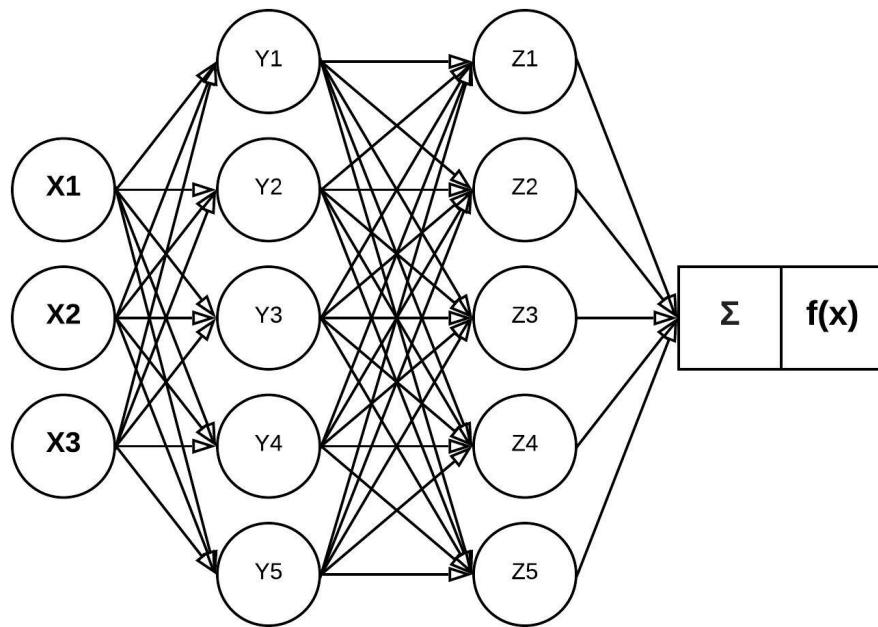
Outro ponto importante de se salientar é que, aqui estamos trabalhando de forma procedural, ou seja, começando dos meios e modelos mais simples para os mais avançados. Por hora, apenas para fins de explicação, estamos trabalhando com poucos neurônios, uma camada oculta, um tipo de função e um tipo de ativação. Porém com uma rápida pesquisa você verá que existem diversos tipos de função de ativação que posteriormente se adequarão melhor a outros modelos conforme nossa necessidade. Nos capítulos subsequentes estaremos trabalhando com modelos diferentes de redes neurais que de acordo com suas particularidades possuem estrutura e número de neurônios e seus respectivos níveis diferentes dos exemplos tradicionais mencionados anteriormente.

Porém tudo ao seu tempo, inicialmente trabalhando com problemas computacionais de menor complexidade os modelos acima são suficientes, posteriormente para problemas mais robustos estaremos entendendo os diferentes modelos e formatos de estrutura de uma rede neural artificial.

Deep Learning

Um dos conceitos mais distorcidos dentro dessa área de redes neurais é o de Deep Learning. Como vimos anteriormente, uma rede neural pode ser algo básico ou complexo de acordo com a necessidade. Para problemas de lógica simples, tomadas de decisão simples ou classificações linearmente separáveis, modelos de perceptron com uma camada oculta junto a uma função de soma e de ativação correta normalmente é o suficiente para solucionar tal problema computacional.

Porém haverão situações onde, para resolução de problemas de maior capacidade teremos de fazer tratamentos para nossos dados de entrada, uso de múltiplas camadas ocultas, além de funções de retroalimentação e ativação específicas para determinados fins. O que acarreta em um volume de processamento de dados muito maior, o que finalmente é caracterizado como deep learning.



A “aprendizagem” profunda se dá quando criamos modelos de rede neural onde múltiplas entradas com seus respectivos pesos passam por múltiplas camadas ocultas, fases supervisionadas ou não, feedforward e backpropagation e por fim algumas horas de processamento. Na literatura é bastante comum também se encontrar este ponto referenciado como redes densas ou redes profundas.

Raciocine que para criarmos uma rede neural que aprende por exemplo, a lógica de uma tabela verdade AND (que inclusive criaremos logo a seguir), estaremos programando poucas linhas de código e o processamento do código se dará em poucos segundos. Agora imagine uma rede neural que identifica e classifica um padrão de imagem de uma mamografia digital em uma

base de dados gigante, além de diversos testes para análise pixel a pixel da imagem para no fim determinar se uma determinada característica na imagem é ou não câncer, e se é ou não benigno ou maligno (criaremos tal modelo posteriormente). Isto sim resultará em uma rede mais robusta onde criaremos mecanismos lógicos que farão a leitura, análise e processamento desses dados, requerendo um nível de processamento computacional alto. Ao final do livro estaremos trabalhando com redes dedicadas a processamento puro de imagens, assim como a geração das mesmas, por hora imagine que este processo envolve múltiplos neurônios, múltiplos nós de comunicação, com múltiplas camadas, etc..., sempre proporcional a complexidade do problema computacional

Então a fim de desmistificar, quando falamos em deep learning, a literatura nos aponta diferentes nortes e ao leigo isso acaba sendo bastante confuso. De forma geral, deep learning será um modelo de rede neural onde teremos múltiplos parâmetros assim como um grande volume de processamento sobre esses dados por meio de múltiplas camadas de processamento.

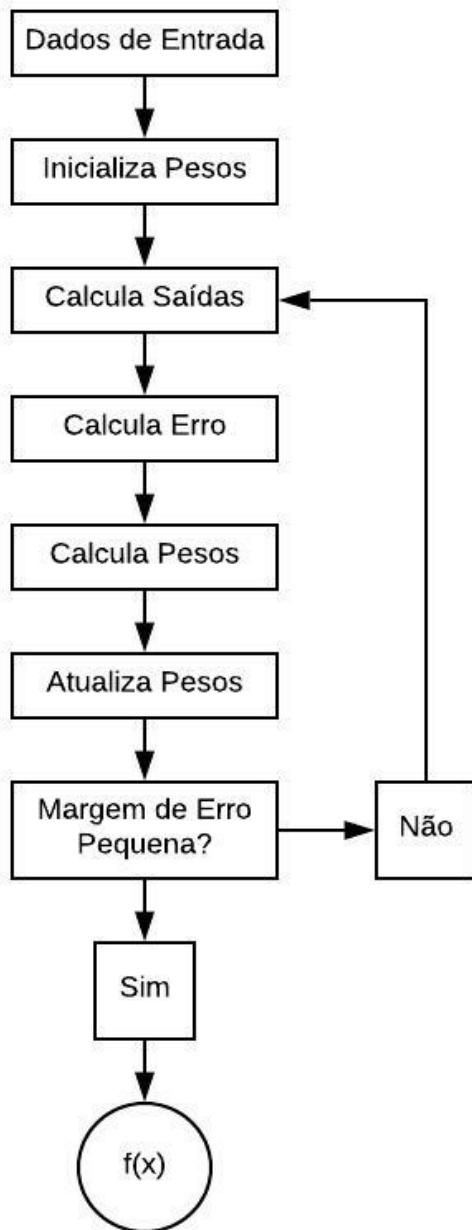
Rotinas de uma Rede Neural Artificial

Independentemente da área que estivermos falando sempre costumamos dividir nossas atribuições em rotinas que executaremos ao longo de um período de tempo, é assim em nossa rotina de trabalho, de estudos, até mesmo de tentar estabelecer padrões de como aproveitar melhor o pouco tempo que temos longe destes... Sendo assim, uma rotina basicamente será aquela sequência de processos que estaremos executando sempre dentro de uma determinada atividade. Em machine learning, no âmbito profissional, teremos diversos tipos de problemas a serem contextualizados para nossas redes neurais e sendo assim teremos uma rotina a ser aplicada sobre esses dados, independente de qual seja o resultado esperado.

No primeiro dataset que usaremos de exemplo para apresentar uma rede neural mais robusta, estaremos executando uma rotina bastante completa que visa extrair o máximo de informações a partir de um banco de dados pré-estabelecido, assim como estaremos criando nossa rede neural em um modelo onde determinados processos serão divididos em blocos que poderão ser reutilizáveis se adaptados para outros contextos posteriormente. Tenha em mente que uma vez criadas nossas ferramentas não há necessidade de cada vez criá-las a partir do zero, estaremos elucidando quais rotinas de processos melhor se adaptam de acordo com os tipos de problema computacional apresentado e dessa forma, uma vez entendido os conceitos e codificadas suas ferramentas, posteriormente você poderá simplesmente reajustar de acordo com a necessidade.

Trabalhando sobre a Breast Cancer Dataset, primeira rede neural mais robusta que estaremos criando e entendendo seus conceitos passo a passo a seguir, estaremos executando uma determinada rotina que não necessariamente se aplica a todos os demais problemas computacionais, mas já servirá de exemplo para criar uma boa bagagem de conhecimento sobre o processo de importação e tratamento dos dados, criação e aplicação de uma rede neural artificial assim como a configuração da mesma a fim de obter melhores resultados. Raciocine que posteriormente em outros datasets estaremos trabalhando e usando outros modelos e ferramentas que não se aplicariam no contexto da Breast Dataset, porém começar por ela promoverá um entendimento bom o suficiente para que você entenda também de acordo com o problema, a rotina de processamento a ser aplicada.

Toda rede neural pode começar a ser contextualizada por um algoritmo bastante simples, onde entre nossas entradas e saídas temos:



Em algumas literaturas ainda podemos encontrar como rotina: Fase de criação e modelagem da rede > Fase supervisionada de reajustes da mesma > Fase de aprendizado de máquina > Fase de testes de performance da rede e dos resultados > Saídas.

Como mencionado nos capítulos anteriores, aqui usaremos de uma abordagem progressiva e bastante didática. Estaremos sim executando esses modelos de rotinas mencionados acima, porém de uma forma mais natural, seguindo a lógica a qual estaremos criando e aplicando linha a linha de código em nossa rede neural.

Sendo assim, partindo para prática estaremos com base inicial no Breast Cancer Dataset entendendo e aplicando cada um dos passos abaixo:

Rotina Breast Cancer Dataset

Rede neural simples:

Importação da base de dados de exemplo

Criação de uma função sigmoide de forma manual

Criação de uma função Sigmoide Derivada de forma manual

Tratamento dos dados, separando em atributos previsores e suas saídas

Criação de uma rede neural simples, camada a camada manualmente com aplicação de pesos aleatórios, seus reajustes em processo de aprendizado de máquina para treino do algoritmo.

Rede neural densa:

Importação da base de dados a partir de arquivo .csv

Tratamento dos dados atribuindo os mesmos em variáveis a serem usadas para rede de forma geral, assim como partes para treino e teste da rede neural.

Criação de uma rede neural densa multicamada que irá inicialmente classificar os dados a partir de uma série de camadas que aplicarão funções pré-definidas e parametrizadas por ferramentas de bibliotecas externas e posteriormente gerar previsões a partir dos mesmos.

Teste de precisão nos resultados e sobre a eficiência do algoritmo de rede neural.

Parametrização manual das ferramentas a fim de obter melhores resultados

Realização de técnicas de validação cruzada, e tuning assim como seus respectivos testes de eficiência.

Teste de viés de confirmação a partir de uma amostra.

Salvar o modelo para reutilização.

Posteriormente iremos trabalhar de forma mais aprofundada em outros modelos técnicas de tratamento e processamento de dados que não se aplicariam corretamente na Breast Cancer Dataset mas a outros tipos de bancos de dados.

Tenha em mente que posteriormente estaremos passo-a-passo trabalhando em modelos onde serão necessários um polimento dos dados no sentido de a partir de um banco de dados bruto remover todos dados faltantes, errôneos ou de alguma forma irrelevantes para o processo.

Estaremos trabalhando com um modelo de abstração chamado de variáveis do tipo Dummy onde teremos um modelo de classificação com devidas particularidades para classificação de muitas saídas.

Também estaremos entendendo como é feito o processamento de imagens por uma rede neural, neste processo, iremos realizar uma série de processos para conversão das informações de pixel a pixel da imagem para dados de uma matriz a ser processado pela rede.

Enfim, sob a sintaxe da linguagem Python e com o auxílio de algumas funções, módulos e bibliotecas estaremos abstraindo e contextualizando problemas da vida real de forma a serem

solucionados de forma computacional por uma rede neural artificial.

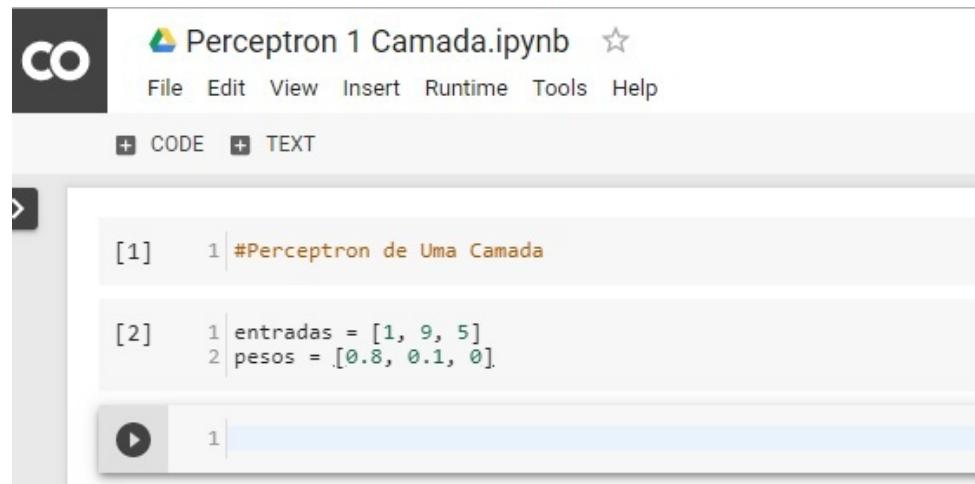
30 – Aprendizado de Máquina

Perceptron de Uma Camada – Modelo Simples

Anteriormente entendemos a lógica de um Perceptron, assim como as particularidades que estes possuem quanto a seu número de camadas, deixando um pouco de lado o referencial teórico e finalmente partindo para prática, a seguir vamos codificar alguns modelos de perceptron, já no capítulo seguinte entraremos de cabeça na codificação de redes neurais artificiais.

Partindo para o Código:

Inicialmente vamos fazer a codificação do algoritmo acima, pondo em prática o mesmo, através do Google Colaboratory (sinta-se livre para usar o Colaboratory ou o Jupyter da suíte Anaconda), ferramentas que nos permitirão tanto criar este perceptron quanto o executar em simultâneo. Posteriormente em problemas que denotarão maior complexidade estaremos usando outras ferramentas.



The screenshot shows the Google Colaboratory interface. At the top, there's a dark header bar with the 'CO' logo, the title 'Perceptron 1 Camada.ipynb' with a star icon, and a file menu: File, Edit, View, Insert, Runtime, Tools, Help. Below the header is a toolbar with 'CODE' and 'TEXT' buttons. The main area contains three code cells. Cell [1] has the comment '#Perceptron de Uma Camada'. Cell [2] contains the variable declarations 'entradas = [1, 9, 5]' and 'pesos = [0.8, 0.1, 0]'. Cell [3] is currently being edited, indicated by a play button icon and the number '1'.

Abrindo nosso Colaboratory inicialmente renomeamos nosso notebook para Perceptron 1 Camada.ipynb apenas por convenção. *Ao criar um notebook Python 3 em seu Colaboratory automaticamente será criada uma pasta em seu Drive de nome Colab Notebooks, onde será salva uma cópia deste arquivo para posterior utilização.

Por parte de código apenas foi criado na primeira célula um comentário, na segunda célula foram declaradas duas variáveis de nomes entradas e pesos, respectivamente. Como estamos atribuindo vários valores para cada uma passamos esses valores em forma de lista, pela sintaxe Python, entre chaves e separados por vírgula. Aqui estamos usando os mesmos valores usados no modelo de perceptron anteriormente descrito, logo, entradas recebe os valores 1, 9 e 5 referentes aos nós de entrada X, Y e Z de nosso modelo assim como pesos recebe os valores 0.8, 0.1 e 0 como no modelo.

```
[3] 1 def soma(e,p):
2     s = 0
3     for i in range(3):
4         s += e[i] * p[i]
5     return s
```

Em seguida criamos uma função de nome soma que recebe como parâmetro as variáveis temporárias e e p. Dentro dessa função inicialmente criamos uma nova variável de nome s que

inicializa com valor 0. Em seguida criamos um laço de repetição que irá percorrer todos valores de e e p, realizar a sua multiplicação e atribuir esse valor a variável s. Por fim apenas deixamos um comando para retornar s, agora com valor atualizado.

```
[4] 1 | s = soma(entradas,pesos)
```

```
[5] 1 | print(s)
```

```
↳ 1.7000000000000002
```

Criamos uma variável de nome s que recebe como atributo a função soma, passando como parâmetro as variáveis entradas e pesos. Executando uma função print() passando como parâmetro a variável s finalmente podemos ver que foram feitas as devidas operações sobre as variáveis, retornando o valor de 1.7, confirmando o modelo anterior.

```
[6] 1 | def stepFunction(s):
2 |   if (s >= 1):
3 |     return 1
4 |   return 0
```

Assim como criamos a função de soma entre nossas entradas e seus respectivos pesos, o processo final desse perceptron é a criação de nossa função degrau. Função essa que simplesmente irá pegar o valor retornado de soma e estimar com base nesse valor se esse neurônio será ativado ou não.

Para isso simplesmente criamos outra função, agora de nome stepFunction() que recebe como parâmetro s. Dentro da função simplesmente criamos uma estrutura condicional onde, se o valor de s for igual ou maior que 1, retornará 1 (referência para ativação), caso contrário, retornará 0 (referência para não ativação).

```
[7] 1 | saida = stepFunction(s)
```

```
[8] 1 | print(saida)
```

```
↳ 1
```

Em seguida criamos uma variável de nome saida que recebe como atributo a função stepFunction() que por sua vez tem como parâmetro s. Por fim executamos uma função print() agora com saida como parâmetro, retornando finalmente o valor 1, resultando como previsto, na ativação deste perceptron.

Código Completo:

```
1 #Perceptron de Uma Camada
2
3 entradas = [1, 9, 5]
4 pesos = [0.8, 0.1, 0]
5
6 def soma(e,p):
7     s = 0
8     for i in range(3):
9         s += e[i] * p[i]
10    return s
11
12 s = soma(entradas,pesos)
13
14 def stepFunction(s):
15     if (s >= 1):
16         return 1
17     return 0
18
19 saida = stepFunction(s)
20
21 print(s)
22 print(saida).
```

```
1.7000000000000002
1
```

Aprimorando o Código:

Como mencionado nos capítulos iniciais, uma das particularidades por qual escolhemos desenvolver nossos códigos em Python é a questão de termos diversas bibliotecas, módulos e extensões que irão facilitar nossa vida oferecendo ferramentas que não só automatizam, mas melhoram processos no que diz respeito a performance. Não existe problema algum em mantermos o código acima usando apenas os recursos do interpretador do Python, porém se podemos realizar a mesma tarefa de forma mais reduzida e com maior performance por quê não o fazer.

Sendo assim, usaremos aplicado ao exemplo atual alguns recursos da biblioteca Numpy, de forma a usar seus recursos internos para tornar nosso código mais enxuto e eficiente. Aqui trabalharemos pressupondo que você já instalou tal biblioteca como foi orientado em um capítulo anterior.

```
[1] 1 import numpy as np
```

Sempre que formos trabalhar com bibliotecas que nativamente não são carregadas e pré-alocadas por nossa IDE precisamos fazer a importação das mesmas. No caso da Numpy, uma vez que essa já está instalada no sistema, basta executarmos o código import numpy para que a mesma seja carregada e possamos usufruir de seus recursos. Por convenção quando importamos alguns tipos de biblioteca ou módulos podemos as referenciar por alguma abreviação, simplesmente para facilitar sua chamada durante o código. Logo, o comando import numpy as np importa a biblioteca Numpy e sempre que a formos usar basta iniciar o código com np.

```
[2] 1 entradas = np.array([1, 9, 5])
2 pesos = np.array([0.8, 0.1, 0])
```

Da mesma forma que fizemos anteriormente, criamos duas variáveis que recebem os respectivos valores de entradas e pesos. Porém, note a primeira grande diferença de código, agora não passamos uma simples lista de valores, agora criamos uma array Numpy para que esses dados sejam vetorizados e trabalhados internamente pelo Numpy. Ao usarmos o comando `np.array()` estamos criando um vetor ou matriz, dependendo da situação, para que as ferramentas internas desta biblioteca possam trabalhar com os mesmos.

```
[3] 1 def Soma(e,p):
2     return e.dot(p)
```

Segunda grande diferença agora é dada em nossa função de Soma, repare que agora ela simplesmente contém como parâmetros `e` e `p`, e ela executa uma única linha de função onde ela retornará o produto escalar de `e` sobre `p`. Em outras palavras, o comando `e.dot(p)` fará o mesmo processo aritmético que fizemos manualmente, de realizar as multiplicações e somas dos valores de `e` com `p`, porém de forma muito mais eficiente.

```
[4] 1 s = Soma(entradas, pesos).
2
3 def stepFunction(Soma):
4     if (s >= 1):
5         return 1
6     return 0
7
8 saida = stepFunction(s)
9
10 print(s)
11 print(saida)
```

```
↳ 1.7000000000000002
1
```

Todo o resto do código é reaproveitado e executado da mesma forma, obtendo inclusive como retorno os mesmos resultados (o que é esperado), a diferença de trocar uma função básica, baseada em listas e condicionais, por um produto escalar realizado por módulo de uma biblioteca dedicada a isto torna o processo mais eficiente. Raciocine que à medida que formos implementando mais linhas de código com mais funções essas pequenas diferenças de performance realmente irão impactar o desempenho de nosso código final.

Código Completo:

```

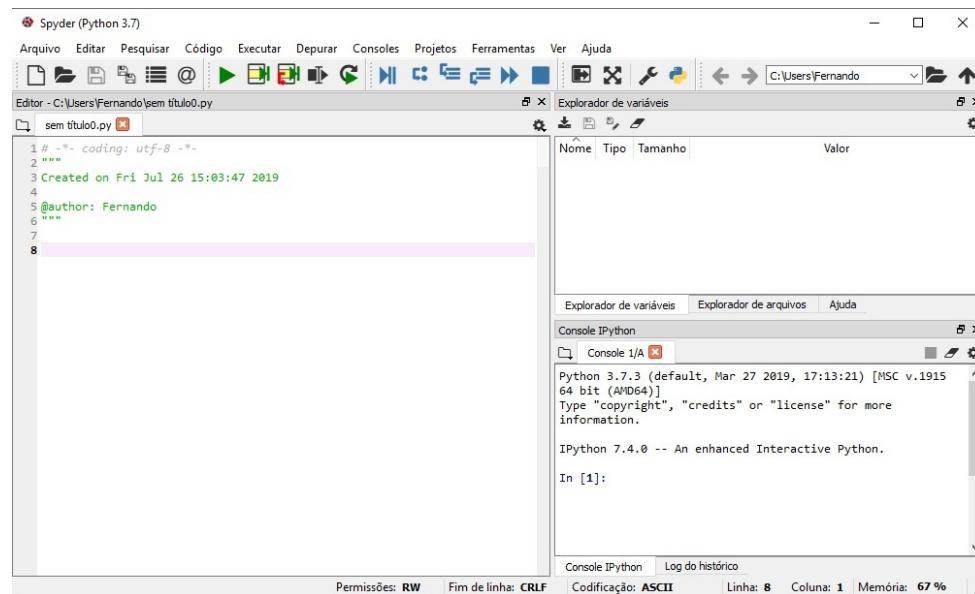
1 import numpy as np
2
3 entradas = np.array([1, 9, 5])
4 pesos = np.array([0.8, 0.1, 0])
5
6 def Soma(e,p):
7     return e.dot(p)
8
9 s = Soma(entradas, pesos)
10
11 def stepFunction(Soma):
12     if (s >= 1):
13         return 1
14     return 0
15
16 saida = stepFunction(s)
17
18 print(s)
19 print(saida)

```

1.7000000000000002
1

Usando o Spyder 3:

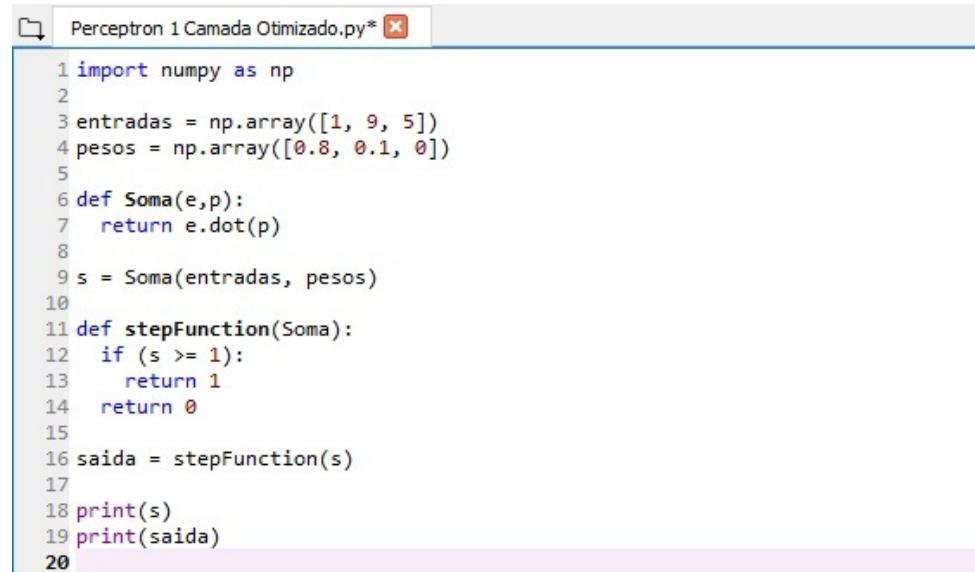
Os códigos apresentados anteriormente foram rodados diretamente no notebook Colab do Google, porém haverão situações de código mais complexo onde não iremos conseguir executar os códigos normalmente a partir de um notebook. Outra ferramenta bastante usada para manipulação de dados em geral é o Spyder. Dentro dele podemos criar e executar os mesmos códigos de uma forma um pouco diferente graças aos seus recursos. Por hora, apenas entenda que será normal você ter de se familiarizar com diferentes ferramentas uma vez que algumas situações irão requerer mais ferramentas específicas. Tratando-se do Spyder, este será o IDE que usaremos para praticamente tudo a partir daqui, graças a versatilidade de por meio dele podermos escrever nossas linhas de código, executá-las individualmente e em tempo real e visualizar os dados de forma mais intuitiva.



Abrindo o Spyder diretamente pelo seu atalho ou por meio da suite Anaconda nos deparamos com sua tela inicial, basicamente o Spyder já vem pré-configurado de forma que possamos simplesmente nos dedicar ao código, talvez a única configuração que você deva fazer é para sua própria comodidade modificar o caminho onde serão salvos os arquivos.

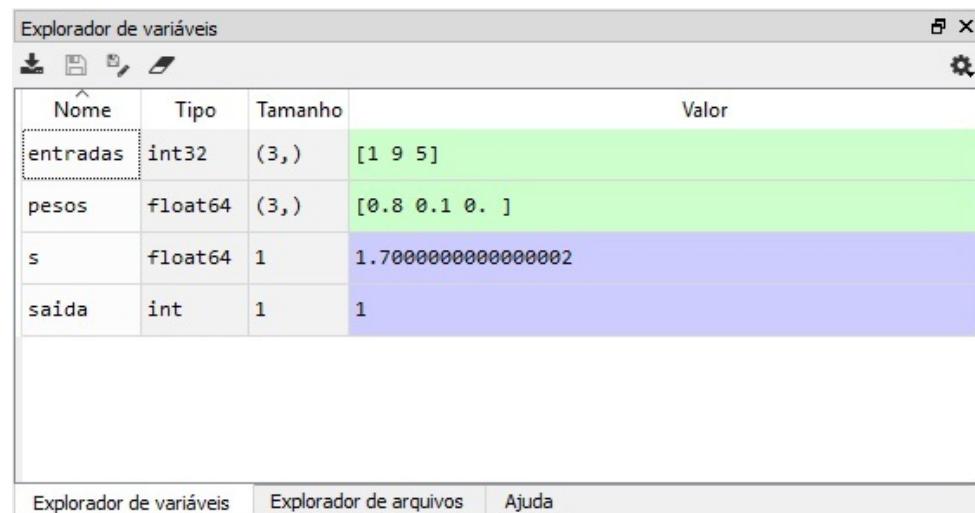
A tela inicial possui a esquerda um espaço dedicado ao código, e a direita um visualizador de variáveis assim como um console que mostra em tempo real certas execuções de blocos de código.

Apenas como exemplo, rodando este mesmo código criado anteriormente, no Spyder podemos em tempo real fazer a análise das operações sobre as variáveis, assim como os resultados das mesmas via terminal.



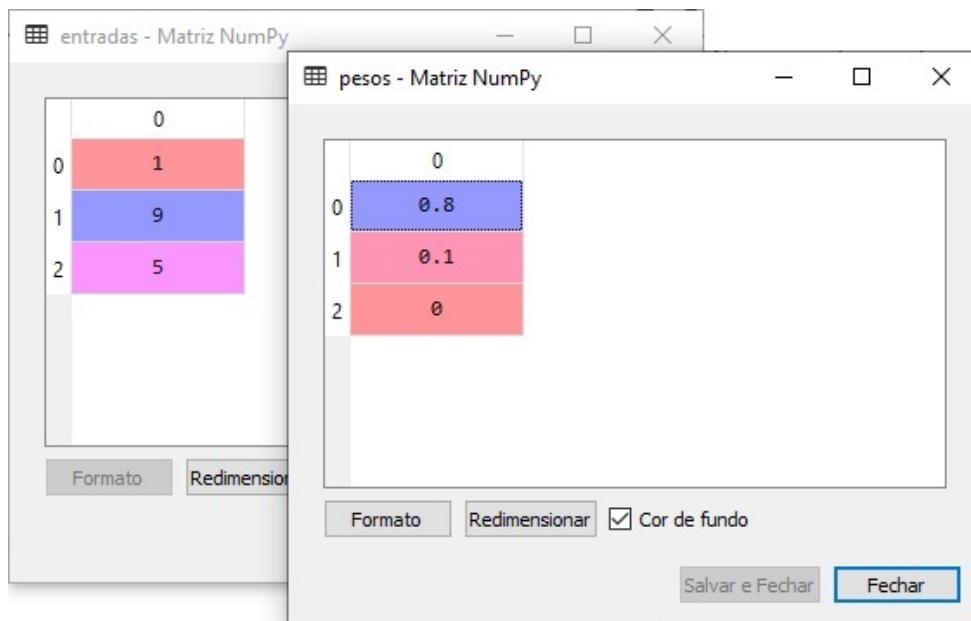
```
Perceptron 1 Camada Otimizado.py*
1 import numpy as np
2
3 entradas = np.array([1, 9, 5])
4 pesos = np.array([0.8, 0.1, 0])
5
6 def Soma(e,p):
7     return e.dot(p)
8
9 s = Soma(entradas, pesos)
10
11 def stepFunction(Soma):
12     if (s >= 1):
13         return 1
14     return 0
15
16 saida = stepFunction(s)
17
18 print(s)
19 print(saida)
20
```

Explorador de Variáveis:



Nome	Tipo	Tamanho	Valor
entradas	int32	(3,)	[1 9 5]
pesos	float64	(3,)	[0.8 0.1 0.]
s	float64	1	1.7000000000000002
saida	int	1	1

Visualizando Variáveis:



Terminal:

```
Console IPython
Console 1/A X
Python 3.7.3 (default, Mar 27 2019, 17:13:21) [MSC v.1915 64 bit (AMD64)
Type "copyright", "credits" or "license" for more information.

IPython 7.4.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/Fernando/Desktop/Livro 2 - Ciéncia de Dados e
Aprendizado de Máquina - Fernando Feltrin/Perceptron 1 Camada Otimizada
wdir='C:/Users/Fernando/Desktop/Livro 2 - Ciéncia de Dados e Aprendizado
Máquina - Fernando Feltrin')
Reloaded modules: colorama, colorama.initialise, colorama.ansitowin32,
colorama.ansi, colorama.winterm, colorama.win32
1.7000000000000002
1

In [2]:
```


Perceptron de Uma Camada – Tabela AND

Entendidos os conceitos lógicos de o que é um perceptron, como os mesmos são um modelo para geração de processamento de dados para por fim ser o parâmetro de uma função de ativação. Hora de, também de forma prática, entender de fato o que é aprendizagem de máquina.

Novamente se tratando de redes neurais serem uma abstração as redes neurais biológicas, o mecanismo de aprendizado que faremos também é baseado em tal modelo. Uma das características de nosso sistema nervoso central é a de aprender criando padrões de conexões neurais para ativação de alguma função ou processamento de funções já realizadas (memória). Também é válido dizer que temos a habilidade de aprender e melhorar nosso desempenho com base em repetição de um determinado tipo de ação.

Da mesma forma, criaremos modelos computacionais de estruturas neurais onde, programaremos determinadas funções de forma manual (fase chamada supervisionada) e posteriormente iremos treinar tal rede para que identifique o que é certo, errado, e memorize este processo. Em suma, a aprendizagem de máquina ocorre quando uma rede neural atinge uma solução generalizada para uma classe de problemas.

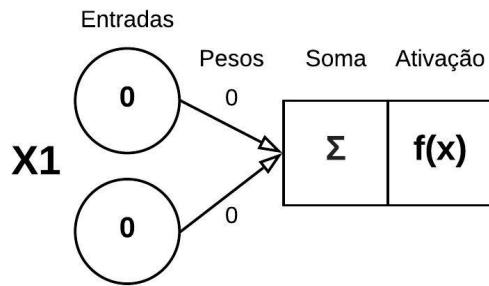
Partindo para prática, vamos criar do zero uma rede neural que aprenderá o que é o mecanismo lógico de uma tabela AND. Operador lógico muito usado em problemas que envolvem simples tomada de decisão. Inicialmente, como é de se esperar, iremos criar uma estrutura lógica que irá processar os valores mas de forma errada, sendo assim, na chamada fase supervisionada, iremos treinar nossa rede para que ela aprenda o padrão referencial correto e de fato solucione o que é o processamento das entradas e as respectivas saídas de uma tabela AND.

Tabela AND:

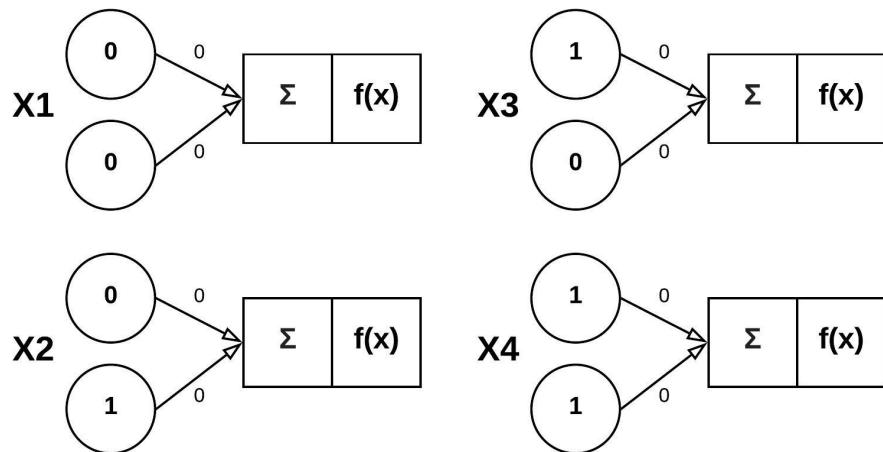
X1	X2	
0	0	0
0	1	0
1	0	0
1	1	1

Todo mundo em algum momento de seu curso de computação teve de estudar operadores lógicos e o mais básico deles é o operador AND. Basicamente ele faz a relação entre duas proposições e apenas retorna VERDADEIRO caso os dois valores de entrada forem verdadeiros. A tabela acima nada mais é do que a tabela AND em operadores aritméticos, mas o mesmo conceito vale para True e False. Sendo 1 / True e 0 / False, apenas teremos como retorno 1 caso os dois operandos forem 1, assim como só será True se as duas proposições forem True.

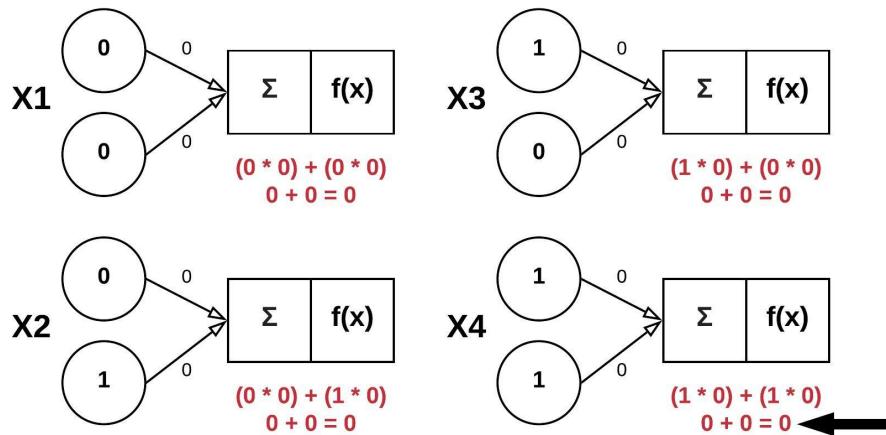
Dessa forma, temos 4 operadores a primeira coluna, mais 4 operadores na segunda camada e na última coluna os resultados das relações entre os mesmos. Logo, temos o suficiente para criar um modelo de rede neural que aprenderá essa lógica.



Basicamente o que faremos é a criação de 4 perceptrons onde cada um terá 2 neurônios de entrada, pesos que iremos atribuir manualmente, uma função soma e uma função de ativação.



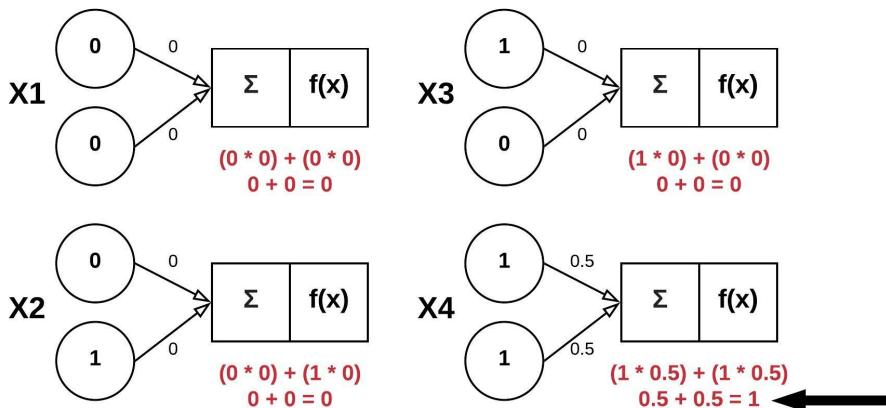
Montada a estrutura visual dos perceptrons, por hora, apenas para fins didáticos, hora de realizar as devidas operações de multiplicação entre os valores de entrada e seus respectivos pesos. Novamente, vale salientar que neste caso em particular, para fins de aprendizado, estamos iniciando essas operações com pesos zerados e definidos manualmente, esta condição não se repetirá em outros exemplos futuros.



Vamos ao modelo, criamos 4 estruturas onde X1 representa a primeira linha de nossa tabela AND (0 e 0), X2 que representa a segunda linha (0 e 1), X3 que representa a terceira linha (1 e 0) e por fim X4 que representa a quarta linha do operador (1 e 1).

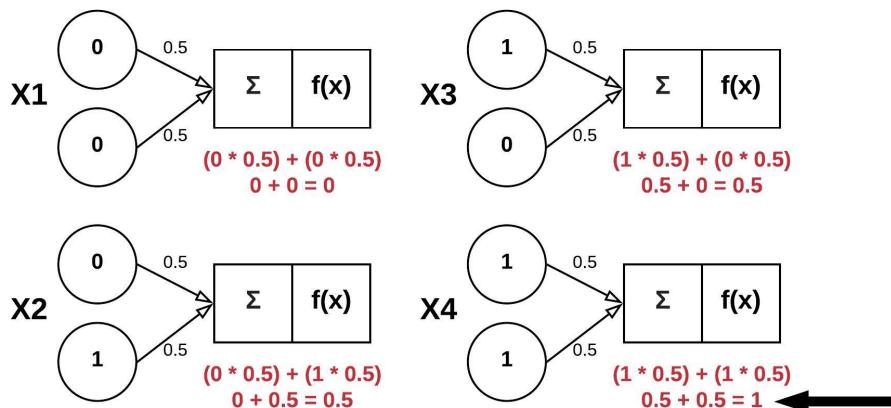
Executando a função de soma dos mesmos, repare que para X1 a multiplicação das entradas pelos respectivos pesos e a soma destes resultou no valor 0, que era esperado nesse caso (Tabela AND - 0 e 0 = 0). O processo se repete exatamente igual aos perceptrons X2 e X3 onde a função soma resulta em 0. Porém, repare em X4, a multiplicação das entradas pelos respectivos pesos como manda a regra resultou em 0, mas sabemos que de acordo com a tabela verdade este resultado deveria ser 1. Isso se deu porque simplesmente os pesos 0 implicaram em uma multiplicação falha, qualquer valor multiplicado por 0 resulta em 0.

Precisamos então treinar nossa rede para que ao final do processo se chegue ao valor esperado ($X4 = 1$). Em redes neurais básicas como estas, o que faremos é o simples reajuste dos valores dos pesos e a repetição do processo para ver se o mesmo é corrigido.



Revisando apenas o perceptron X4, alterando os valores dos pesos de 0 para 0.5 e repetindo a função de soma, agora sim temos o valor esperado 1. (Tabela AND 1 e 1 = 1).

Uma vez encontrado o valor de pesos que solucionou o problema a nível de X4, hora de aplicar este mesmo valor de pesos a toda rede neural e verificar se este valor funciona para tudo.



Aplicando o valor de peso 0.5 a todos neurônios e refeitos os devidos cálculos dentro de nossa função soma, finalmente temos os valores corrigidos. Agora seguindo a lógica explicada anteriormente, como os valores de X2 e X3 são menores que 1, na step function os mesmos serão reduzidos a 0, sendo apenas X4 o perceptron ativado nessa rede. o que era esperado de acordo com a tabela AND.

Em X1 0 e 0 = 0, em X2 0 e 1 = 0, em X3 1 e 0 = 0 e em X4 1 e 1 = 1.

Partindo para o Código:

```
[1] 1 import numpy as np
```

Por parte do código, todo processo sempre se inicia com a importação das bibliotecas e módulos necessários para que possamos fazer o uso deles. Nesse caso todas as operações que faremos serão operações nativas do Python ou funções internas da biblioteca Numpy. Para sua importação basta executar o comando import numpy, por convenção também as referenciaremos como np por meio do comando as, dessa forma, sempre que quisermos “chamar” alguma função sua basta usar np.

```
[2] 1 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
2 saídas = np.array([0,0,0,1])
3 pesos = np.array([0,0,0,0])
```

Em seguida criamos três variáveis que ficarão responsáveis por guardar os valores das entradas, saídas e dos pesos. Para isso declaramos uma nova variável de nome entradas que recebe como atributo uma array numpy através do comando np.array que por sua vez recebe como parâmetros os valores de entrada. Repare na sintaxe e também que para cada item dessa lista estão contidos os valores X1 e X2, ou seja, cada linha de nossa tabela AND. Da mesma forma criamos uma variável saídas que recebe uma array numpy com uma lista dos valores de saída. Por fim criamos uma variável pesos que recebe uma array numpy de valores iniciais zerados. Uma vez que aqui neste exemplo estamos explorando o que de fato é o aprendizado de máquina, usaremos pesos inicialmente zerados que serão corrigidos posteriormente.

```
[3] 1 | taxaAprendizado = 0.5
```

Logo após criamos uma variável de nome taxaAprendizado que como próprio nome sugere, será a taxa com que os valores serão atualizados e testados novamente nesse perceptron. Você pode testar valores diferentes, menores, e posteriormente acompanhar o processo de aprendizagem.

```
[4] 1 | def Soma(e,p):  
2 |     return e.dot(p).
```

Na sequência criamos a nossa função Soma, que tem como parâmetro as variáveis temporárias e e p, internamente ela simplesmente tem o comando de retornar o produto escalar de e sobre p, em outras palavras, a soma das multiplicações dos valores atribuídos as entradas com seus respectivos pesos.

```
[5] 1 | s = Soma(entradas, pesos).
```

Criamos uma variável de nome s que como atributo recebe a função Soma que agora finalmente possui como parâmetro as variáveis entradas e pesos. Dessa forma, podemos instanciar a variável s em qualquer bloco de código que ela imediatamente executará a soma das entradas pelos pesos, guardando seus dados internamente.

```
[6] 1 | def stepFunction(soma):  
2 |     if (soma >= 1):  
3 |         return 1  
4 |     return 0
```

Seguindo com o código, criamos nossa stepFunction, em tradução livre, nossa função degrau, aquela que pegará os valores processados para consequentemente determinar a ativação deste perceptron ou não em meio a rede. Para isso definimos uma função stepFunction que tem como parâmetro a variável temporária soma. Internamente criamos uma estrutura condicional básica onde, se soma (o valor atribuído a ela) for igual ou maior que 1, retorna 1 (resultando na ativação), caso contrário retornará 0 (resultando na não ativação do mesmo).

```
[7] 1 | def calculoSaida(reg):  
2 |     s = reg.dot(pesos)  
3 |     return stepFunction(s)
```

Assim como as outras funções, é preciso criar também uma função que basicamente pegará o valor processado e atribuído como saída e fará a comparação com o valor que já sabemos de saída da tabela AND, para que assim se possa identificar onde está o erro, subsequentemente realizar as devidas modificações para se aprender qual os valores corretos de saída. Simplesmente criamos uma função, agora de nome calculoSaida que tem como parâmetro a variável temporária reg, internamente ela tem uma variável temporária s que recebe como atributo o produto escalar de reg sobre pesos, retornando esse valor processado pela função degrau.

```
[8] 1 def aprendeAtualiza():
2     erroTotal = 1
3     while (erroTotal != 0):
4         erroTotal = 0
5         for i in range (len(saidas)):
6             calcSaida = calculoSaida(np.array(entradas[i]))
7             erro = abs(saidas[i] - calcSaida)
8             erroTotal += erro
9             for j in range(len(pesos)):
10                 pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
11             print('Pesos Atualizados> ' + str(pesos[j]))
12     print('Total de Erros: ' +str(erroTotal))
```

Muita atenção a este bloco de código, aqui será realizado o processo de aprendizado de máquina propriamente dito.

Inicialmente criamos uma função de nome aprendeAtualiza, sem parâmetros mesmo. Internamente inicialmente criamos uma variável de nome erroTotal, com valor 1 atribuído simbolizando que existe um erro a ser corrigido.

Em seguida criamos um laço de repetição que inicialmente tem como parâmetro a condição de que se erroTotal for diferente de 0 será executado o bloco de código abaixo. Dentro desse laço erroTotal tem seu valor zerado, na sequência uma variável i percorre todos valores de saídas, assim como uma nova variável calcSaida recebe os valores de entrada em forma de array numpy, como atributo de calculoSaida. É criada também uma variável de nome erro que de forma absoluta (sem considerar o sinal negativo) recebe como valor os dados de saídas menos os de calcSaida, em outras palavras, estamos fazendo nesta linha de código a diferença entre os valores de saída reais que já conhecemos na tabela AND e os que foram encontrados inicialmente pelo perceptron. Para finalizar esse bloco de código erroTotal soma a si próprio o valor encontrado e atribuído a erro.

Na sequência é criado um novo laço de repetição onde uma variável j irá percorrer todos valores de pesos, em sequida pesos recebe como atributo seus próprios valores multiplicados pelos de taxaAprendizado (definido manualmente anteriormente como 0.5), multiplicado pelos valores de entradas e de erro. Para finalizar ele recebe dois comandos print() para que no console/terminal sejam exibidos os valores dos pesos atualizados assim como o total de erros encontrados.

```
[9] 1 | aprendeAtualiza()
```

```
↳ Pesos Atualizados> 0.0
Pesos Atualizados> 0.5
Pesos Atualizados> 0.5
Total de Erros: 1
Pesos Atualizados> 0.5
Total de Erros: 0
```

Por fim simplesmente executando a função aprendeAtualiza podemos acompanhar via console/terminal o processo de atualização dos pesos (aprendizado de máquina) e o log de quando não existem mais erros, indicando que tal perceptron finalmente foi treinado e aprendeu de fato a interpretar uma tabela AND, ou em outras palavras, reconhecer o padrão correto de suas entradas e saídas.

Código Completo:

```

[1] 1 import numpy as np
2
3 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
4 saidas = np.array([0,0,0,1])
5 pesos = np.array([0.0,0.0])
6
7 taxaAprendizado = 0.5
8
9 def Soma(e,p):
10     return e.dot(p)
11
12 s = Soma(entradas, pesos)
13
14 def stepFunction(soma):
15     if (soma >= 1):
16         return 1
17     return 0
18
19 def calculoSaida(reg):
20     s = reg.dot(pesos)
21     return stepFunction(s)
22
23 def aprendeAtualiza():
24     erroTotal = 1
25     while (erroTotal != 0):
26         erroTotal = 0
27         for i in range (len(saidas)):
28             calcSaida = calculoSaida(np.array(entradas[i]))
29             erro = abs(saidas[i] - calcSaida)
30             erroTotal += erro
31             for j in range(len(pesos)):
32                 pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
33                 print('Pesos Atualizados> ' + str(pesos[j]))
34             print('Total de Erros: ' +str(erroTotal))
35
36 aprendeAtualiza()

```

Usando Spyder:

```

1 import numpy as np
2
3 entradas = np.array([[0,0],[0,1],[1,0],[1,1]])
4 saidas = np.array([0,0,0,1])
5 pesos = np.array([0.0,0.0])
6
7 taxaAprendizado = 0.5
8
9 def Soma(e,p):
10     return e.dot(p)
11
12 s = Soma(entradas, pesos)
13
14 def stepFunction(soma):
15     if (soma >= 1):
16         return 1
17     return 0
18
19 def calculoSaida(reg):
20     s = reg.dot(pesos)
21     return stepFunction(s)
22
23 def aprendeAtualiza():
24     erroTotal = 1
25     while (erroTotal != 0):
26         erroTotal = 0
27         for i in range (len(saidas)):
28             calcSaida = calculoSaida(np.array(entradas[i]))
29             erro = abs(saidas[i] - calcSaida)
30             erroTotal += erro
31             for j in range(len(pesos)):
32                 pesos[j] = pesos[j] + (taxaAprendizado * entradas[i][j] * erro)
33                 print('Pesos Atualizados > ' + str(pesos[j]))
34             print('Total de Erros: ' +str(erroTotal))
35
36 aprendeAtualiza()

```

Explorador de Variáveis:

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]]
pesos	float64	(2,)	[0.5 0.5]
s	float64	(4,)	[0. 0. 0. 0.]
saidas	int32	(4,)	[0 0 0 1]
taxaAprendizado	float	1	0.5

Console:

Perceptron Multicamada – Tabela XOR

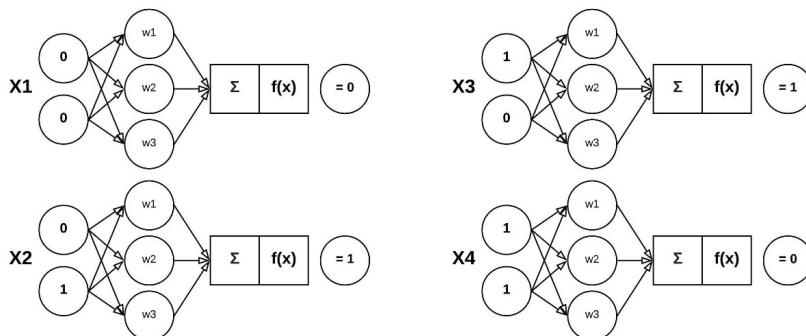
Entendida a lógica do processo de aprendizado de máquina na prática, treinando um perceptron para que aprenda um operador lógico AND, hora de aumentar levemente a complexidade. Como dito anteriormente, gradualmente vamos construindo essa bagagem de conhecimento. O principal diferencial deste capítulo em relação ao anterior é que, em uma tabela AND temos um problema linearmente separável, onde a resposta (saída) era basicamente 1 ou 0, pegando o exemplo de uma tabela XOR temos o diferencial de que este operador lógico realiza a operação lógica entre dois operandos, que resulta em um valor lógico verdadeiro se e somente se o número de operandos com valor verdadeiro for ímpar, dessa forma temos um problema computacional onde, a nível de perceptron, teremos de treinar o mesmo para descobrir a probabilidade desta operação ser verdadeira.

Em outras palavras, teremos de identificar e treinar nosso perceptron para que realize mais de uma camada de correção de pesos, afim de identificar a probabilidade correta da relação entre as entradas e as saídas da tabela XOR que por sua vez possui mais de um parâmetro de ativação. O fato de trabalharmos a partir daqui com uma camada a mais de processamento para rede, camada essa normalmente chamada de camada oculta, permite que sejam aplicados “filtros” que nos ajudam a identificar, treinar e aprender sob nuances de valores muito menores do que os valores brutos do exemplo anterior.

Tabela XOR:

X1	X2	
0	0	0
0	1	1
1	0	1
1	1	0

Estrutura Lógica:



Diretamente ao código:

```
Perceptron Multicamada XOR.py
1 import numpy as np
2
```

Como de costume, todo processo se inicia com a importação das bibliotecas e módulos que utilizaremos ao longo de nossa rede. Agora trabalhando diretamente com o Spyder, podemos em tempo real executar linhas ou blocos de código simplesmente os selecionando com o mouse e aplicando o comando Ctrl + ENTER. Para importação da biblioteca Numpy basta executar o comando import como exemplificado acima.

```
3 entradas = np.array([[0,0],
4                 [0,1],
5                 [1,0],
6                 [1,1]])
7 saidas = np.array([[0],[1],[1],[0]])
```

Em seguida criamos as variáveis entradas e saídas que como seus nomes já sugerem, recebem como atributos os respectivos dados da tabela XOR, em forma de array numpy, assim, uma vez vetorizados, poderemos posteriormente aplicar funções internas desta biblioteca.

Explorador de variáveis				
Nome	Tipo	Tamanho	Valor	
entradas	int32	(4, 2)	[[0 0] [0 1]]	
saidas	int32	(4, 1)	[[0] [1]]	

Executando esse bloco de código (via Ctrl + ENTER) podemos ver que ao lado direito da interface, no Explorador de variáveis são criadas as respectivas variáveis.



Clicando duas vezes sobre as mesmas podemos explorar de forma visual, em forma de matrizes como previsto, lado a lado compondo a tabela XOR.

```
9 pesos0 = np.array([[-0.424, -0.740, -0.961],
10                [0.358, -0.577, -0.469]])
11 pesos1 = np.array([[-0.017], [-0.893], [0.148]])
```

Em seguida criamos as variáveis dedicadas a guardar os valores dos pesos, aqui, novamente apenas para fins de exemplo, estamos iniciando essas variáveis com valores aleatórios.

Nome	Tipo	Tamanho	Valor
entradas	int32	(4, 2)	[[0 0] [0 1]]
pesos0	float64	(2, 3)	[[-0.424 -0.74 -0.961] [0.358 -0.577 -0.469]]
pesos1	float64	(3, 1)	[[-0.017] [-0.893]]
saidas	int32	(4, 1)	[[0] [1]]

Se a sintaxe estiver correta, ao selecionar e executar este bloco de código podemos ver no explorador de variáveis que tais variáveis foram corretamente criadas.

```
13 ntreinos = 100
14 taxaAprendizado = 0.3
15 momentum = 1
```

Em seguida criamos estas três variáveis auxiliares que nos serão úteis posteriormente. A variável ntreinos, que aqui recebe como atributo o valor 100, diz respeito ao parâmetro de quantas vezes a rede será executada para que haja o devido ajuste dos pesos. Da mesma forma a variável taxaAprendizado aqui com valor 0.3 atribuído é o parâmetro para de quantos em quantos números os pesos serão modificados durante o processo de aprendizado, esse parâmetro, entendido em outras literaturas como a velocidade em que o algoritmo irá aprender, conforme o valor pode inclusive fazer com que o algoritmo piore sua eficiência ou entre em um loop onde fica estagnado em um ponto, logo, é um parâmetro a ser testado com outros valores e comparar a eficiência dos resultados dos mesmos. Por fim momentum é um parâmetro padrão, uma constante na fórmula de correção da margem de erro, nesse processo de atualização de pesos, aqui, para este exemplo, segue com o valor padrão 1 atribuído, mas pode ser testado com outros valores para verificar se as últimas alterações surtiram melhoria da eficiência do algoritmo.

```
17 def sigmoid(soma):
18     return 1 / (1 + np.exp(-soma))
```

Logo após criamos nossa Função Sigmoide, que neste modelo, substitui a Função Degrau que utilizamos anteriormente. Existem diferentes funções de ativação, as mais comuns, Degrau e Sigmoide utilizaremos com frequência ao longo dos demais exemplos. Em suma, a grande diferença entre elas é que a Função Degrau retorna valores absolutos 0 ou 1 (ou valores definidos pelo usuário, mas sempre neste padrão binário, um ou outro), enquanto a Função Sigmoide leva em consideração todos valores intermediários entre 0 e 1, dessa forma, conseguimos verificar a probabilidade de um dado valor se aproximar de 0 ou se aproximar de 1, de acordo com suas características.

Para criar a função Sigmoide, simplesmente definimos sigmoid que recebe como parâmetro a variável temporária soma, internamente ela simplesmente retorna o valor 1 dividido pela soma de 1 pela exponenciação de soma, o que na prática nos retornará um valor float entre 0 e 1 (ou seja, um número com casas decimais).

```
20 for i in range(ntreinos):
21     camadaEntrada = entradas
22     somaSinapse0 = np.dot(camadaEntrada, pesos0)
23     camadaOculta = sigmoid(somaSinapse0)
24
25     somaSinapse1 = np.dot(camadaOculta, pesos1)
26     camadaSaida = sigmoid(somaSinapse1)
27
28     erroCamadaSaida = saidas - camadaSaida
29     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
```

Prosseguindo criamos o primeiro bloco de código onde de fato ocorre a interação entre as variáveis. Criamos um laço de repetição que de acordo com o valor setado em ntreinos executa as seguintes linhas de código. Inicialmente é criada uma variável temporária camadaEntrada que recebe como atributo o conteúdo de entradas, em seguida é criada uma variável somaSinapse0, que recebe como atributo o produto escalar (soma das multiplicações) de camadaEntrada por pesos0. Apenas relembrando, uma sinapse é a termologia da conexão entre neurônios, aqui, trata-se de uma variável que faz a interligação entre os nós da camada de entrada e da camada oculta. Em seguida é criada uma variável camadaOculta que recebe como atributo a função sigmoid que por sua vez tem como parâmetro o valor resultante dessa operação sobre somaSinapse0.

Da mesma forma é necessário criar a estrutura de interação entre a camada oculta e a camada de saída. Para isso criamos a variável somaSinapse1 que recebe como atributo o produto escalar entre camadaOculta e pesos1, assim como anteriormente fizemos, é criada uma variável camadaSaida que por sua vez recebe como atributo o valor gerado pela função sigmoid sobre somaSinapse1.

Por fim são criados por convenção duas variáveis que nos mostrarão parâmetros para avaliar a eficiência do processamento de nossa rede. erroCamadaSaida aplica a fórmula de erro, fazendo a simples diferença entre os valores de saídas (valores que já conhecemos) e camadaSaida (valores encontrados pela rede). Posteriormente criamos a variável mediaAbsoluta que simplesmente, por meio da função `.mean()` transforma os dados de erroCamadaSaida em um valor percentual (Quantos % de erro existe sobre nosso processamento).

Nome	Tipo	Tamanho	Valor
camadaEntrada	int32	(4, 2)	[[0 0] [0 1]]
camadaOculta	float64	(4, 3)	[[0.5 0.5 0.5] [0.5885562 0.35962319 0.38485296 ...]]
camadaSaida	float64	(4, 1)	[[0.40588573] [0.43187857]]
entradas	int32	(4, 2)	[[0 0] [0 1]]
erroCamadaSaida	float64	(4, 1)	[[-0.40588573] [0.56812143]]
i	int	1	99
mediaAbsoluta	float64	1	0.49880848923713045

Selecionando e executando esse bloco de código são criadas as referentes variáveis, das camadas de processamento assim como as variáveis auxiliares. Podemos clicando duas vezes sobre as mesmas visualizar seu conteúdo. Por hora, repare que neste primeiro processamento é criada a variável mediaAbsoluta que possui valor 0.49, ou seja 49% de erro em nosso processamento, o que é uma margem muito grande. Pode ficar tranquilo que valores altos nesta etapa de execução são perfeitamente normais, o que faremos na sequência é justamente trabalhar a realizar o aprendizado de máquina, fazer com que nossa rede identifique os padrões corretos e reduza esta margem de erro ao menor valor possível.

```
31 def sigmoideDerivada(sig):
32     return sig * (1-sig)
33 sigDerivada = sigmoid(0.5)
34 sigDerivada1 = sigmoideDerivada(sigDerivada)
```

Dando sequência criamos nossa função `sigmoideDerivada` que como próprio nome sugere, faz o cálculo da derivada, que também nos será útil na fase de aprendizado de máquina. Basicamente o

que fazemos é definir uma função sigmoideDerivada que tem como parâmetro a variável temporária sig. Internamente é chamada a função que retorna o valor obtido pela multiplicação de sig (do valor que estiver atribuído a esta variável) pela multiplicação de 1 menos o próprio valor de sig. Em seguida, aproveitando o mesmo bloco de código dedicado a esta etapa, criamos duas variáveis sigDerivada, uma basicamente aplica a função sigmoid com o valor de 0.5, a outra, aplica a própria função sigmoideDerivada sobre sigDerivada.

```
36 derivadaSaida = sigmoideDerivada(camadaSaida)
37 deltaSaida = erroCamadaSaida * derivadaSaida
```

Da mesma forma, seguimos criando mais variáveis que nos auxiliarão a acompanhar o processo de aprendizado de máquina, dessa vez criamos uma variável derivadaSaida que aplica a função sigmoideDerivada para a camadaSaida, por fim criamos a variável deltaSaida que recebe como atributo o valor da multiplicação entre erroCamadaSaida e derivadaSaida. Vale lembrar que o que nomeamos de delta em uma rede neural normalmente se refere a um parâmetro usado como referência para correto ajuste da descida do gradiente. Em outras palavras, para se realizar um ajuste fino dos pesos e consequentemente conseguir minimizar a margem de erro dos mesmos, são feitos internamente uma série de cálculos para que esses reajustes sejam feitos da maneira correta, parâmetros errados podem prejudicar o reconhecimento dos padrões corretos por parte da rede neural.

	0
0	0.241143
1	0.245359
2	0.246004
3	0.248237

	0
0	-0.0978763
1	0.139394
2	0.138553
3	-0.113696

Podemos sempre que quisermos, fazer a visualização das variáveis por meio do explorador, aqui, os dados encontrados executando a fórmula do cálculo da derivada sobre os valores de saída assim como os valores encontrados para o delta, em outras palavras, pelo sinal do delta podemos entender como (em que direção no grafo) serão feitos os reajustes dos pesos, uma vez que individualmente eles poder ter seus valores aumentados ou diminuídos de acordo com o processo.

```
39 pesos1Transposta = pesos1.T
40 deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41 deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
```

Dando sequência, teremos de fazer um reajuste de formado de nossos dados em suas devidas matrizes para que as operações aritméticas sobre as mesmas possam ser realizadas corretamente. Raciocine que quando estamos trabalhando com vetores e matrizes temos um padrão de linhas e

colunas que pode ou não ser multiplicável. Aqui nesta fase do processo teremos inconsistência de formatos de nossas matrizes de pesos, sendo necessário realizar um processo chamado Matriz Transposta, onde transformaremos linhas em colunas e vice versa de forma que possamos realizar as operações e obter valores corretos.

Inicialmente criamos uma variável pesos1Transposta que recebe como atributo pesos1 com aplicação da função .T, função interna da Numpy que realizará essa conversão. Em seguida criamos uma variável de nome deltaSaidaXpesos que recebe como atributo o produto escalar de deltaSaida por pesos1Transposta. Por fim aplicamos novamente a fórmula do delta, dessa vez para camada oculta, multiplicando deltaSaidaXpesos pelo resultado da função sigmoide sob o valor de camadaOculta.

	0	1	2
0	-0.017	-0.893	0.148
1			
2			

Apenas visualizando a diferença entre pesos1 e pesos1Transposta (colunas transformadas em linhas).

Se você se lembra do algoritmo explicado em capítulos anteriores verá que até este momento estamos trabalhando na construção da estrutura desse perceptron, assim como sua alimentação com valores para seus nós e pesos. A esta altura, obtidos todos valores iniciais, inclusive identificando que tais valores resultam em erro se comparado ao esperado na tabela XOR, é hora de darmos início ao que para alguns autores é chamado de backpropagation, ou seja, realizar de fato os reajustes dos pesos e o reprocessamento do perceptron, repetindo esse processo até o treinar de fato para a resposta correta. Todo processo realizado até o momento é normalmente tratado na literatura como feed forward, ou seja, partindo dos nós de entrada fomos alimentando e processando em sentido a saída, a ativação desse perceptron, a etapa de backpropagation como o nome já sugere, retroalimenta o perceptron, faz o caminho inverso ao feed forward, ou em certos casos, retorna ao ponto inicial e refaz o processamento a partir deste.

O processo de backpropagation inclusive tem uma fórmula a ser aplicada para que possamos entender a lógica desse processo como operações sobre nossas variáveis.

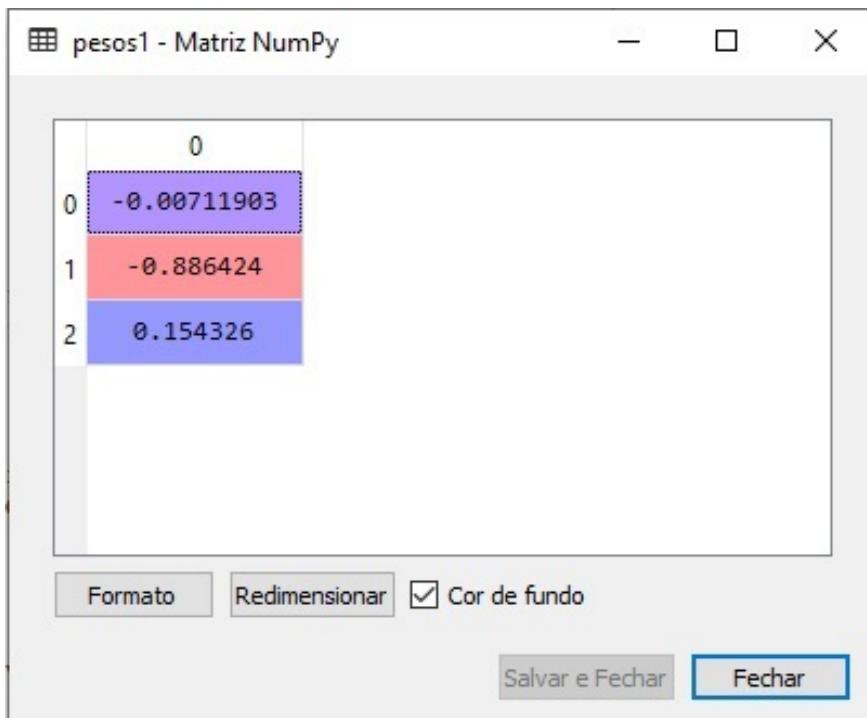
$$\text{peso}(n + 1) = (\text{peso}(n) + \text{momento}) + (\text{entrada} * \text{delta} * \text{taxa de aprendizagem})$$

```

43 camadaOcultaTransposta = camadaOculta.T
44 pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45 pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)

```

Da mesma forma como fizemos anteriormente, criamos uma variável camadaOcultaTransposta que simplesmente recebe a camadaOculta de forma transposta por meio da função .T. Logo após usamos a variável pesos3 que recebe como atributo o produto escalar de camadaOcultaTransposta pelo deltaSaida. Finalmente, fazemos o processo de atualização dos valores de pesos1, atribuindo ao mesmo os valores atualizados de pesos1 multiplicado pelo momentum somado com os valores de pesos3 multiplicados pelo parâmetro de taxaAprendizado.



Executando esse bloco de código você pode ver que de fato houve a atualização dos valores de pesos1. Se nesse processo não houver nenhuma inconsistência ou erro de sintaxe, após a execução desses blocos de código temos a devida atualização dos pesos da camada oculta para a camada de saída.

```

47 camadaEntradaTransposta = camadaEntrada.T
48 pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49 pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)

```

O mesmo processo é realizado para atualização dos valores de pesos0. Ou seja, agora pela lógica backpropagation voltamos mais um passo ao início do perceptron, para que possamos fazer as devidas atualizações a partir da camada de entrada.

```

25 for i in range(ntreinos):
26     camadaEntrada = entradas
27     somaSinapse0 = np.dot(camadaEntrada, pesos0)
28     camadaOculta = sigmoid(somaSinapse0)
29
30     somaSinapse1 = np.dot(camadaOculta, pesos1)
31     camadaSaida = sigmoid(somaSinapse1)
32
33     erroCamadaSaida = saidas - camadaSaida
34     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
35
36     derivadaSaida = sigmoideDerivada(camadaSaida)
37     deltaSaida = erroCamadaSaida * derivadaSaida
38
39     pesos1Transposta = pesos1.T
40     deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41     deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
42
43     camadaOcultaTransposta = camadaOculta.T
44     pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45     pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
46
47     camadaEntradaTransposta = camadaEntrada.T
48     pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49     pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)

```

Lembrando que Python é uma linguagem interpretada e de forte indentação, o que em outras palavras significa que o interpretador lê e executa linha após linha em sua sequência normal, e executa blocos de código quando estão uns dentro dos outros de acordo com sua indentação. Sendo assim, todo código praticamente pronto, apenas reorganizamos o mesmo para que não haja problema de interpretação. Dessa forma, todos blocos dedicados aos ajustes dos pesos ficam dentro daquele nosso laço de repetição, para que possamos executá-los de acordo com os parâmetros que iremos definir para as épocas e taxa de aprendizado.

```
51     print('Margem de Erro: ' +str(mediaAbsoluta))
```

Finalmente criamos uma função `print()` dedicada a nos mostrar via console a margem de erro deste processamento. Agora selecionando e executando todo o código, podemos acompanhar via console o aprendizado de máquina acontecendo. Lembrando que inicialmente havíamos definido o número de vezes a serem executado o código inicialmente como 100. Esse parâmetro pode ser livremente modificado na variável `ntreinos`.

Console IPython

Console 1/A

```
Margem de Erro: 0.49767112522643897
Margem de Erro: 0.4976484214145201
Margem de Erro: 0.4976254998572073
Margem de Erro: 0.4976023592551371
Margem de Erro: 0.4975789982549841
Margem de Erro: 0.49755541545037174
Margem de Erro: 0.4975316093827602
Margem de Erro: 0.49750757854230654
Margem de Erro: 0.49748332136870266
Margem de Erro: 0.4974588362519857
Margem de Erro: 0.4974341215333255
Margem de Erro: 0.4974091755057867
Margem de Erro: 0.4973839964150665
Margem de Erro: 0.49735858246020803
Margem de Erro: 0.4973329317942916
Margem de Erro: 0.4973070425251006
```

In [30]:

Executando 100 vezes a margem de erro cai para 0.497.

Console IPython

Console 1/A

```
Margem de Erro: 0.36869231728556695
Margem de Erro: 0.3685977889255375
Margem de Erro: 0.3685034121821234
Margem de Erro: 0.36840918664731676
Margem de Erro: 0.3683151119144694
Margem de Erro: 0.36822118757829
Margem de Erro: 0.3681274132348392
Margem de Erro: 0.3680337884815269
Margem de Erro: 0.3679403129171078
Margem de Erro: 0.36784698614167816
Margem de Erro: 0.3677538077566716
Margem de Erro: 0.36766077736485536
Margem de Erro: 0.3675678945703269
Margem de Erro: 0.36747515897850924
Margem de Erro: 0.36738257019614784
Margem de Erro: 0.3672901278313063
```

In [31]:

Executando 1000 vezes, a margem de erro cai para 0.367.

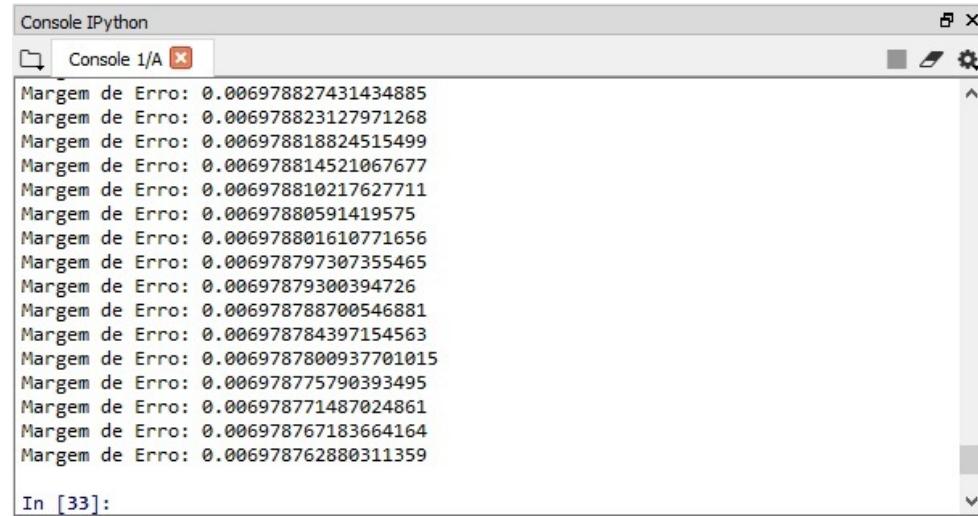
Console IPython

Console 1/A

```
Margem de Erro: 0.13312615898616278
Margem de Erro: 0.1331261170573894
Margem de Erro: 0.13312607512928198
Margem de Erro: 0.1331260332018406
Margem de Erro: 0.13312599127506525
Margem de Erro: 0.13312594934895583
Margem de Erro: 0.1331259074235124
Margem de Erro: 0.13312586549873495
Margem de Erro: 0.1331258235746234
Margem de Erro: 0.13312578165117778
Margem de Erro: 0.133125739728398
Margem de Erro: 0.13312569780628414
Margem de Erro: 0.1331256558848362
Margem de Erro: 0.13312561396405417
Margem de Erro: 0.1331255720439378
Margem de Erro: 0.1331255301244873
```

In [32]:

Executando 100000 vezes, a margem de erro cai para 0.133.



```
Margem de Erro: 0.006978827431434885
Margem de Erro: 0.006978823127971268
Margem de Erro: 0.006978818824515499
Margem de Erro: 0.006978814521067677
Margem de Erro: 0.006978810217627711
Margem de Erro: 0.00697880591419575
Margem de Erro: 0.006978801610771656
Margem de Erro: 0.006978797307355465
Margem de Erro: 0.00697879300394726
Margem de Erro: 0.006978788700546881
Margem de Erro: 0.006978784397154563
Margem de Erro: 0.0069787800937701015
Margem de Erro: 0.006978775790393495
Margem de Erro: 0.006978771487024861
Margem de Erro: 0.006978767183664164
Margem de Erro: 0.006978762880311359
```

In [33]:

Por fim, rede executada 1000000 de vezes, a margem de erro cai para 0.006, em outras palavras, agora a rede está treinada para identificar os padrões de entrada e saída de uma tabela XOR, com 0,006% de margem de erro (ou 99,994% de precisão). Lembrando que em nossa amostragem inicial tínhamos 51% de acerto, agora quase 100% de acerto é um resultado excelente.

Código Completo:

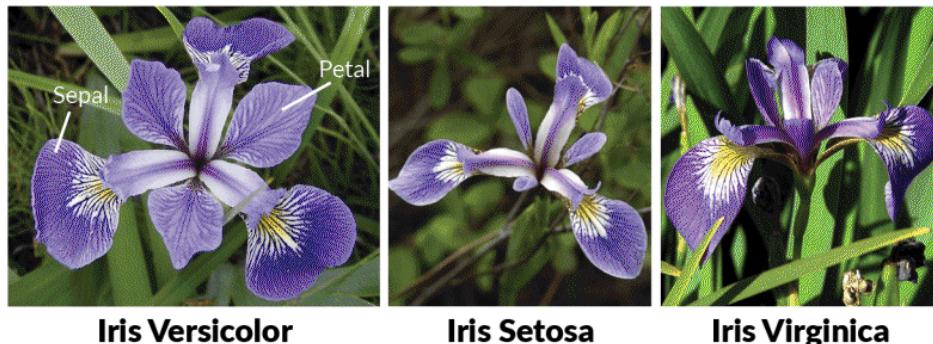
```
1 import numpy as np
2
3 entradas = np.array([[0,0],
4                      [0,1],
5                      [1,0],
6                      [1,1]])
7 saidas = np.array([[0],[1],[1],[0]])
8
9 pesos0 = np.array([[-0.424, -0.740, -0.961],
10                     [0.358, -0.577, -0.469]])
11 pesos1 = np.array([[-0.017], [-0.893], [0.148]])
12
13 ntreinos = 1000000
14 taxaAprendizado = 0.3
15 momentum = 1
16
17 def sigmoid(soma):
18     return 1 / (1 + np.exp(-soma))
19
20 def sigmoideDerivada(sig):
21     return sig * (1-sig)
22 sigDerivada = sigmoid(0.5)
23 sigDerivada1 = sigmoideDerivada(sigDerivada)
```

```
25 for i in range(ntreinos):
26     camadaEntrada = entradas
27     somaSinapse0 = np.dot(camadaEntrada, pesos0)
28     camadaOculta = sigmoid(somaSinapse0)
29
30     somaSinapse1 = np.dot(camadaOculta, pesos1)
31     camadaSaida = sigmoid(somaSinapse1)
32
33     erroCamadaSaida = saidas - camadaSaida
34     mediaAbsoluta = np.mean(np.abs(erroCamadaSaida))
35
36     derivadaSaida = sigmoideDerivada(camadaSaida)
37     deltaSaida = erroCamadaSaida * derivadaSaida
38
39     pesos1Transposta = pesos1.T
40     deltaSaidaXpesos = deltaSaida.dot(pesos1Transposta)
41     deltaCamadaOculta = deltaSaidaXpesos * sigmoideDerivada(camadaOculta)
42
43     camadaOcultaTransposta = camadaOculta.T
44     pesos3 = camadaOcultaTransposta.dot(deltaSaida)
45     pesos1 = (pesos1 * momentum) + (pesos3 * taxaAprendizado)
46
47     camadaEntradaTransposta = camadaEntrada.T
48     pesos4 = camadaEntradaTransposta.dot(deltaCamadaOculta)
49     pesos0 = (pesos0 * momentum) + (pesos4 * taxaAprendizado)
50
51     print('Margem de Erro: ' +str(mediaAbsoluta))
```

31 – Redes Neurais Artificiais

Classificação Multiclasse – Aprendizado de Máquina Simples – Iris Dataset

Em 1936, o estatístico e biólogo britânico Ronald Fisher publicou seu artigo intitulado *The use of multiple measurements in taxonomic problems*, que em tradução livre seria algo como O Uso de Múltiplas Medições em Problemas Taxonômicos, onde publicou seu estudo classificatório das plantas do tipo Íris quanto às características dos formatos de suas pétalas e sépalas em relação ao seu tamanho.



Fazendo o uso desse artigo científico como base, Edgar Anderson coletou e quantificou os dados para finalmente classificar estas amostras em 3 tipos de flores diferentes a partir de métodos computacionais estatísticos publicando em 2012 o seu estudo *An Approach for Iris Plant Classification Using Neural Network*, que em tradução livre seria algo como Uma Abordagem para Classificação das Plantas Íris Usando Redes Neurais.

O método de classificação utilizado em ambos estudos foi igualmente dividir o conjunto em grupos de amostras e separá-las de acordo com o comprimento e a largura em centímetros das pétalas e das sépalas das mesmas, já que este era um padrão entre aqueles tipos de flores na natureza. Vale lembrar que a metodologia de classificação dessas amostras em sua época foi manual e estatística, agora o que faremos é aplicar um modelo computacional para classificação das mesmas e, até mais importante que isso, criar um modelo que possa ser aplicado para novas classificações.

Por fim, para termos em prática os métodos de classificação de dados utilizaremos da Iris Dataset, que nada mais é do que um dos conjuntos de dados inclusos na biblioteca SKLearn, baseado no trabalho de Ronald Fisher. *Tal modelo pode ser aplicado para diferentes situações e tipos de dados, aqui utilizaremos o Iris Dataset para inicialmente entendermos as mecânicas usadas quando temos de classificar amostras de vários tipos.

Inicialmente faremos uma abordagem mais simples aplicando alguns modelos classificatórios e no capítulo subsequente estaremos aplicando uma metodologia baseada em rede neural artificial densa.

Como já comentado anteriormente, estaremos aprendendo de forma procedural, quanto mais formos avançando nossos estudos, mais prática será a abordagem.

Partindo para o Código:

```
1 from sklearn.datasets import load_iris  
2
```

Como sempre, todo processo se inicia com a importação das bibliotecas e módulos que serão utilizados ao longo do código e que não fazem parte das bibliotecas pré-alocadas da IDE. Aqui inicialmente iremos utilizar a Iris Dataset que está integrada na biblioteca SKLearn para fins de estudo. Quando estamos importando uma biblioteca inteira simplesmente executamos o comando import seguido do nome da biblioteca, aqui o que faremos é importar parte do conteúdo de uma biblioteca, é bastante comum importarmos algum módulo ou função no lugar de toda uma biblioteca. Por meio do comando from seguido de sklearn.datasets seguido de import load_iris, estamos importando a sub biblioteca de exemplo load_iris, do módulo datasets da biblioteca sklearn. Selecione e executando esse código (Ctrl + ENTER) se não houve nenhum erro de sintaxe você simplesmente não terá nenhuma mensagem no console, caso haja algum traceback significa que houve algum erro no processo de importação.

```
3 base = load_iris()  
4
```

Em seguida criamos uma variável de nome base que recebe como atributo todo conteúdo contido em load_iris. Sem parâmetros mesmo.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module

Dando uma olhada no Explorador de Variáveis, é possível ver que foi criada uma variável reconhecida como objeto parte da biblioteca sklearn.

base - Dicionário (6 elementos)

Chave	Tipo	Tamanho	Valor
DESCR	str	1	... _iris_dataset: [
data	float64	(150, 4)	[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
feature_names	list	4	['sepal length (cm)', 'sepal width (cm)', 'length (cm)', 'petal ...']
filename	str	1	C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\datasets\data\ir ...
target	int32	(150,)	[0 0 0 ... 2 2 2]
target_names	str320	(3,)	ndarray object of numpy module

Clicando duas vezes sobre a variável é possível explorar a mesma de forma visual. Assim é possível reconhecer que existem 4 colunas (Chave, Tipo, Tamanho e Valor) assim como 6 linhas (DESCR, data, feature_names, filename, target e target_names).

Uma das etapas que será comum a praticamente todos os exemplos que iremos utilizar é uma fase onde iremos fazer o polimento desses dados, descartando os que não nos interessam e reajustando os que usaremos de forma a poder aplicar operações aritméticas sobre os mesmos. Podemos aplicar essa etapa diretamente na variável ou reaproveitar partes do conteúdo da mesma em outras variáveis. Aqui, inicialmente faremos o reaproveitamento, em exemplos futuros trabalharemos aplicando esse polimento diretamente sobre o banco de dados inicial.

```
5 print(base.data)
6 print(base.target)
7 print(base.target_names)
```

Podemos visualizar tais dados de forma manual por meio da função `.print()`, dessa forma, teremos a exibição do conteúdo em sua forma normal no console/terminal.

```
...: print(base.data)
[[5.1 3.5 1.4 0.2]
 [4.9 3. 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5. 3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5. 3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3. 1.4 0.1]
 [4.3 3. 1.1 0.1]
 [5.8 4. 1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]]
```

Retorno obtido em `print(base.data)`, aqui se encontram os dados das medições em forma de uma matriz com 150 amostras/registros.

```
In [ ]: print(base.target)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Retorno obtido em `print(base.target)`, onde é possível ver que para cada uma das 150 amostras anteriores existe um valor referente a sua classificação indo de 0 a 2.

```
In [ ]: print(base.target_names)
['setosa' 'versicolor' 'virginica']
```

Retorno obtido em `print(base.target_names)`, onde podemos relacionar a numeração anterior 0, 1 e 2 agora com os nomes dos três tipos de plantas, 0 para setosa, 1 para versicolor e 2 para virginica.

```
9 entradas = base.data
10 saidas = base.target
11 rotulos = base.target_names
```

Em seguida criamos três variáveis, entradas que recebe como atributo os dados contidos apenas em data da variável base. saidas que recebe como atributo os dados de target da variável base e por fim uma variável de nome rotulos que recebe o conteúdo de target_names da variável base.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
rotulos	str320	(3,)	ndarray object of numpy module
saidas	int32	(150,)	[0 0 0 ... 2 2 2]

Se todo processo correu normalmente no explorador de variáveis é possível visualizar tanto as variáveis quanto seu conteúdo.

The screenshot shows the Jupyter Notebook's Variable Explorer interface with four data frames:

- entradas - Matriz NumPy**: A 6x4 matrix of floating-point numbers representing four features for 150 samples. The data is as follows:

	0	1	2	3
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4

- saidas - Matriz NumPy**: A 6x1 vector of integers representing the target values for each sample, all set to 0.
- rotulos - Matriz NumPy**: A 3x1 vector of categorical labels: 'setosa' at index 0, 'versicolor' at index 1, and 'virginica' at index 2.
- base**: A Bunch object of the sklearn.utils module.

Relacionando os dados, em entradas temos os quatro atributos que foram coletados para cada uma das 150 amostras (tamanho e comprimento das pétalas e das sépalas). Em saídas temos a relação de que, de acordo com os parâmetros contidos em entradas tal planta será classificada em um dos três tipos, 0, 1 e 2. Por fim, de acordo com essa numeração podemos finalmente saber de acordo com a classificação qual o nome da planta.

```
13 print(base.data.shape)
14 print(base.target.shape)
```

Por meio do comando `.shape` podemos de fato verificar o tamanho dessas matrizes de dados para que possamos fazer o cruzamento e/ou aplicar funções aritméticas sobre os mesmos.

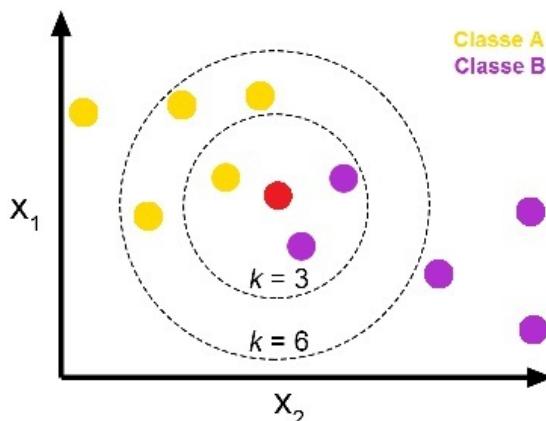
```
In [ ]: print(base.data.shape)
...: print(base.target.shape)
(150, 4)
(150,)
```

Selecionando e executando esse bloco de código podemos verificar via terminal que data tem um formato de 150 linhas e 4 colunas, assim como saídas tem 150 linhas e uma coluna.

Terminada a fase de polimento dos dados hora de começarmos a finalmente processar os mesmos para transformá-los de dados brutos para informação estatística.

Aplicando o Modelo KNN

Agora iremos por em prática a análise estatística dos nossos dados, para isto começaremos usando o modelo KNN, que do inglês K Nearest Neighbors, em tradução livre significa Vizinhos Próximos de K. Neste tipo de modelo estatístico teremos um item de verificação por convenção chamado K e o interpretador com base nos dados fornecidos fará a leitura e exibirá como resultado os dados mais próximos, que mais se assemelham a K. Em outras palavras, definiremos um parâmetro, onde de acordo com a proximidade (ou distância) do valor da amostra para esse parâmetro, conseguimos obter uma classificação.



Repare no gráfico, o ponto vermelho representa K, o número/parâmetro que definiremos para análise KNN, próximo a K existem 3 outros pontos, um referente a classe A e dois referentes a classe B. O modelo estatístico KNN trabalhará sempre desta forma, a partir de um número K ele fará a classificação dos dados quanto a proximidade a ele.

Pela documentação do KNN podemos verificar que existe uma série de etapas a serem seguidas para aplicação correta deste modelo:

1º - Pegamos uma série de dados não classificados.

2º - O interpretador irá mensurar a proximidade desses dados quanto a sua semelhança.

3º - Definimos um número K, para que seja feita a análise deste e de seus vizinhos.

4º - Aplicamos os métodos KNN.

5º - Analisamos e processamos os resultados.

```
16 from sklearn.neighbors import KNeighborsClassifier  
17
```

Para aplicar o modelo estatístico KNN não precisamos criar a estrutura lógica mencionada acima do zero, na verdade iremos usar tal classificador que já vem pronto e pré-configurado no módulo neighbors da biblioteca sklearn. Assim como fizemos no início de nosso código, iremos por meio do comando from importar apenas o KneighborsClassifier de dentro de sklearn, muita atenção a sintaxe do código, para importação correta você deve respeitar as letras maiúsculas e minúsculas da nomenclatura.

```
18 knn = KNeighborsClassifier(n_neighbors = 1)  
19 knn.fit(entradas, saídas)  
20 knn.predict([[5.1,3.1,1.4,0.2]])
```

Logo após criamos uma variável de nome knn que recebe como atributo KneighborsClassifier, que por sua vez tem o parâmetro n_neighbors = 1, o que significa que tais dados serão classificados quanto a proximidade de 1 número em relação a sua amostra vizinha. Este parâmetro, inclusive, pode ser alterado para realização de outros testes de performance posteriormente. Em seguida aplicamos sobre knn a função .fit(), essa função por sua vez serve para alimentar o classificador com os dados a serem processados. Repare que knn.fit recebe como parâmetro todos dados de entradas e saídas. Por fim, também podemos fazer uma simples previsão por meio da função .predict(), aqui, pegando os dados de uma linha de nossa base de dados podemos realizar uma previsão sobre ela.

```
In [ ]: from sklearn.neighbors import KNeighborsClassifier  
...:  
...: knn = KNeighborsClassifier(n_neighbors = 1)  
...: knn.fit(entradas, saídas)  
...: knn.predict([[5.1,3.1,1.4,0.2]])  
Out[ ]: array([0])
```

Após selecionar todo bloco de código anterior e executá-lo, acompanhando via console podemos ver que a criação do classificador foi feita de maneira correta assim como a alimentação do mesmo com os valores contidos em entradas e saídas. Via console também é possível ver o retorno da função de previsão sobre os valores retirados de uma das linhas da matriz base. O retorno por hora foi um simples “0” o que de imediato não significa muita coisa. Será comum você ter de apresentar seus dados e suas conclusões, seja para fins de estudo ou profissionais, e mostrar um zero pode não significar nada para quem não entende do assunto. Para um resultado mais claro vamos associar este resultado inicial com os rótulos que separamos lá no início desse código.

```
22 especie = knn.predict([[5.9,3.5,1.8]])[0]  
23 print(especie)  
24 rotulos[especie]
```

Para isso criamos uma variável de nome especie que recebe como atributo nosso classificador knn e a função .predict() que por sua vez recebe como parâmetro outra linha da matriz escolhida aleatoriamente. Note que ao final do código de atribuição existe um parâmetro em forma de posição de lista [0]. Executando uma função print() sobre especie, e associando o conteúdo da mesma com o de rotulos como retorno deveremos ter o rótulo associado a posição 0 da lista de espécies.

```
In [  ]: especie = knn.predict([[5.9,3.5,1.8]])[0]
...: print(especie)
...: rotulos[especie]
2
Out[  ]: 'virginica'
```

De fato, é o que acontece, agora como resultado das associações temos como retorno ‘virginica’ o que é uma informação muito mais relevante do que ‘0’.

Seguindo com a análise dos dados, um método bastante comum de ser aplicado é o de treino e teste do algoritmo, para que de fato se possa avaliar a performance do modelo. Na prática imagine que você tem uma amostra a ser classificada, feita toda a primeira etapa anterior, uma maneira de obter um viés de confirmação é pegar novamente a base de dados, dividir ela em partes e as testar individualmente, assim é possível comparar e avaliar a performance com base nos testes entre si e nos testes em relação ao resultado anterior. Todo processo pode ser realizado por uma ferramenta dedicada a isso da biblioteca sklearn, novamente, tal ferramenta já vem pré-configurada para uso, bastando fazer a importação, a alimentação e a execução da mesma.

```
26 from sklearn.model_selection import train_test_split
27
```

Executando o comando acima, respeitando a sintaxe, importamos a ferramenta `train_test_split` do módulo `model_selection` da biblioteca `sklearn`.

```
28 etreino, eteste, streino, steste = train_test_split(entradas,
29                                     saidas,
30                                     test_size = 0.25)
```

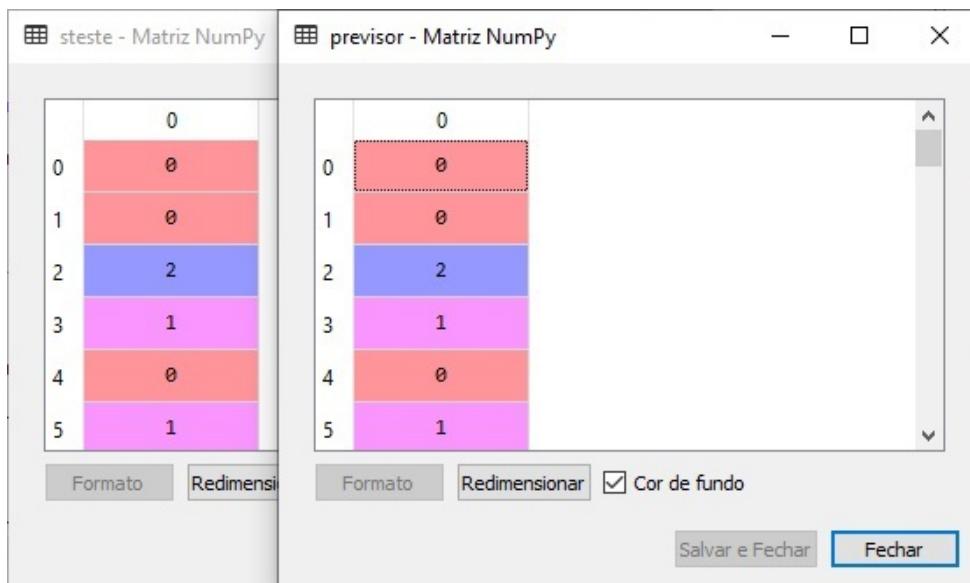
A seguir criamos quatro variáveis dedicadas a conter amostras dos dados para treino e para teste do modelo, por convenção as chamamos de `entreino` (entradas para treino), `eteste` (entradas para teste), `streino` (saídas para treino) e `steste` (saídas para teste). Todas elas recebem como atributo a função `train_test_split` que por sua vez recebe como parâmetro os dados contidos em `entradas`, `saídas` e um terceiro parâmetro chamado `test_size`, aqui definido como `0.25`, onde 25% das amostras serão dedicadas ao treino, consequentemente 75% das amostras serão usadas para teste. Você pode alterar esse valor livremente, porém é notado que se obtém melhores resultados quando as amostras não são divididas igualmente (50%/50%), lembrando também que uma base de treino muito pequena pode diminuir a precisão dos resultados.

```
31 knn.fit(entreino, streino)
32 previsor = knn.predict(eteste)
```

Em seguida usamos novamente nossa variável de nome `knn` que já executa sobre si a função `.fit()` que por sua vez recebe como parâmetro os dados contidos em `entreino` e `streino`. Dando sequência criamos uma variável de nome `previsor` que executa como atributo a função `knn.predict` passando como parâmetro para a mesma os dados contidos em `eteste`.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	6	Bunch object of sklearn.utils module
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
especie	int32	1	2
eteste	float64	(38, 4)	[[4.7 3.2 1.3 0.2] [5.2 3.4 1.4 0.2]]
etreino	float64	(112, 4)	[[6.2 2.2 4.5 1.5] [6.1 2.6 5.6 1.4]]
previsor	int32	(38,)	[0 0 2 ... 2 0 1]
rotulos	str320	(3,)	ndarray object of numpy module

Selecionando e executando todo bloco de código acima é possível ver que as devidas variáveis foram criadas.



Selecionando as variáveis com dados de saída é possível fazer a comparação entre elas e avaliar de forma visual se os valores encontrados em ambos os testes condizem. É normal haver disparidade entre esses dados, e isso é uma consequência normal de termos uma base de dados pequena, e a reduzir ainda mais no processo de dividir para treino e teste. Raciocine que aqui nossa base de dados contém apenas 150 amostras, dependendo do contexto você terá bases de dados de centenas de milhares de amostras/registros, além de seus atributos. Quanto menor for a base de dados, menos é a precisão na taxa de acertos para previsões.

Além de dividir as amostras e as testar individualmente, outro processo bastante comum é usar ferramentas que de fato avaliam a performance de nossos modelos, uma das mais comuns a ser usada é a ferramenta `accuracy_score` da biblioteca `sklearn`.

```
34 from sklearn import metrics
35
```

Um pouco diferente das importações realizadas anteriormente, aqui não temos como importar apenas a ferramenta em questão, ela depende de outras funções contidas no mesmo pacote em seu módulo, sendo necessário importar todo o módulo do qual ela faz parte. Para isso simplesmente importamos o módulo `metrics` da biblioteca `sklearn` por meio do código acima.

```

36 margem_acertos = metrics.accuracy_score(steste, previsor)
37

```

Prosseguindo com o código, criamos uma nova variável de nome margem_acertos que recebe como atributo a execução de metrics.accuracy_score, que por sua vez recebe como parâmetro os dados de steste e de previsor.

Nome	Tipo	Tamanho	Valor
etreino	float64	(112, 4)	[[6.2 2.2 4.5 1.5] [6.1 2.6 5.6 1.4]]
margem_acertos	float64	1	0.9473684210526315
previsor	int32	(38,)	[0 0 2 ... 2 0 1]

Selecionando e executando o bloco de código anterior é criada a variável margem_acertos, que nos mostra de forma mais clara um valor de 0.947, o que em outras palavras significa que o resultado das comparações dos valores entre steste e previsores tem uma margem de acerto de 94%.

Outra metodologia bastante comum é, retornando ao KNN, alterar aquele parâmetro n_neighbors definido anteriormente como 1 e realizar novos testes, inclusive é possível criar um laço de repetição que pode testar e exibir resultados com vários parâmetros dentro de um intervalo.

```

38 valores_k = {}
39 k=1
40 while k < 25:
41     knn = KNeighborsClassifier(n_neighbors=k)
42     knn.fit(etreino, streino)
43     previsores_k = knn.predict(eteste)
44     acertos = metrics.accuracy_score(steste, previsores_k)
45     valores_k[k] = acertos
46     k += 1

```

Aqui vamos diretamente ao bloco de código inteiro, supondo que você domina a lógica de laços de repetição em Python. Inicialmente o que fazemos é criar uma variável valores_k que recebe como atributo uma chave vazia. Em seguida definimos que o valor inicial de k é 1. Sendo assim, criamos uma estrutura de laço de repetição onde, enquanto o valor de k for menor que 25 é executado o seguinte bloco de código: Em primeiro lugar nossa variável knn recebe o classificador mas agora seu parâmetro não é mais 1, mas o valor que estiver atribuído a k. Em seguida knn é alimentada pela função .fit() com os dados contidos em etreino e streino. Logo após criamos uma variável previsores_k que recebe a função previsora sob os dados contidos em eteste. Na sequência criamos uma variável de nome acertos que realiza o teste de previsão sobre os valores contidos em steste e previsores_k. Sendo assim valores_k recebe cada valor atribuído a k e replica esse valor em acertos. Por fim k é incrementado em 1 e todo laço se repete até que k seja igual a 25.

valores_k - Dicionário (24 elementos)

Chave	Tipo	Tamanho	Valor
1	float64	1	0.9473684210526315
2	float64	1	0.9473684210526315
3	float64	1	0.9473684210526315
4	float64	1	0.9473684210526315
5	float64	1	0.9736842105263158
6	float64	1	0.9736842105263158
7	float64	1	1.0
8	float64	1	1.0
9	float64	1	1.0
10	float64	1	1.0
11	float64	1	1.0

Salvar e Fechar Fechar

Selecionando e executando todo esse código é criada a variável valores_k onde podemos visualizar que foram realizados testes, cada um com um valor de k diferente, entre 1 e 25 com sua respectiva margem de acerto. Dessa forma, para essa pequena base de dados, podemos ver que vários valores diferentes de k tem uma altíssima margem de acertos, em bases maiores ou com dados mais heterogêneos, é mais comum que esses valores de k sejam bastante divergentes entre si. Aqui nesse exemplo o valor mais baixo encontrado foi de 94% de precisão mesmo.

Outra situação bastante comum é, uma vez que temos informações relevantes a partir de nossos dados, criar mecanismos para que os mesmos possam ser exibidos de forma mais clara. Uma das coisas mais comuns em ciência de dados é apresentar os dados em forma de gráfico. Para isso também usaremos uma ferramenta de fácil integração e uso em nossa IDE, chamada matplotlib.

```
48 import matplotlib.pyplot as plt
49
```

A importação dessa biblioteca segue os mesmos moldes usados anteriormente, por meio do comando import e podemos referenciá-la como plt por comodidade.

```
50 plt.plot(list(valores_k.keys()),
51           list(valores_k.values()))
```

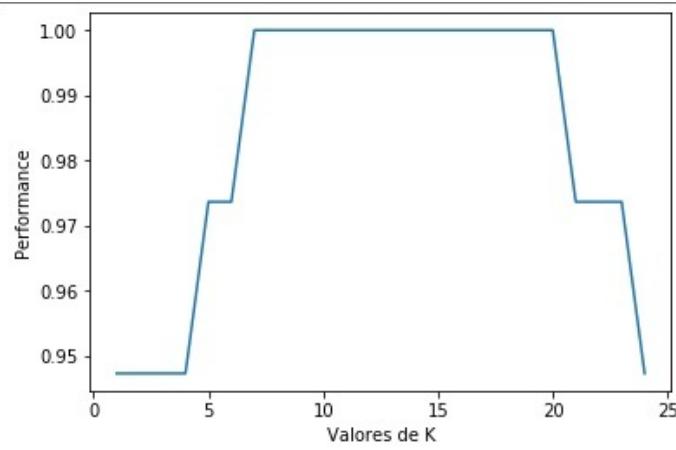
Raciocine que esta plotagem em gráfico é uma forma simples e visual de apresentar os dados via console/terminal. Sendo assim, primeiramente criamos uma função plt.plot() que recebe como atributos, em forma de lista, as chaves e os valores de valores_k, respectivamente as referenciando por .keys() e por .values(). Assim podemos ter um plano cartesiano com eixos X e Y onde tais dados serão exibidos.

```
52 plt.xlabel('Valores de K')
53 plt.ylabel('Performance')
```

Inclusive uma prática comum é definir rótulos para os eixos X e Y do gráfico para que fique de ainda mais fácil interpretação.

```
54 plt.show()  
55
```

Executando a função plt.show() o gráfico será gerado e exibido em nosso console.



Analisando o gráfico podemos ver que no eixo X (horizontal) existe a relação dos valores de k no intervalo entre 1 e 25. Já no eixo Y (vertical) podemos notar que foram exibidos valores entre 95% e 100% de margem de acerto.

Apenas por curiosidade, muito provavelmente você replicou as linhas de código exatamente iguais as minhas e seu gráfico é diferente, note que a cada execução do mesmo código o gráfico será diferente porque a cada execução do mesmo ele é reprocessado gerando novos valores para todas as variáveis.

Aplicando Regressão Logística

Outra prática bastante comum que também pode ser aplicada sobre nosso modelo é a chamada Regressão Logística, método bastante usado para realizar previsões categóricas. Ou seja, testar amostras para descobrir qual a probabilidade de a mesma fazer parte de um tipo ou de outro.

```
56 from sklearn.linear_model import LogisticRegression  
57
```

Como sempre, o processo se inicia com a importação do que for necessário para nosso código, nesse caso, importamos a ferramenta LogisticRegression do módulo linear_model da biblioteca sklearn por meio do comando acima.

```
58 regl = LogisticRegression()  
59 regl.fit(etreino, streino)
```

Inicialmente criamos uma variável de nome regl que recebe como atributo o regressor LogisticRegression, sem parâmetros mesmo. Em seguida já o alimentamos com os dados de etreino e streino por meio da função .fit().

```

Out[ ]:
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, l1_ratio=None, max_iter=100,
                    multi_class='warn', n_jobs=None, penalty='l2',
                    random_state=None, solver='warn', tol=0.0001, verbose=0,
                    warm_start=False)

```

Repare em seu console que, como nesse caso, quando criamos uma variável e atribuímos a ela alguma ferramenta, sem passar nenhum parâmetro, a mesma é criada normalmente com seus parâmetros pré-configurados, se no código não haviam parâmetros não significa que tal ferramenta internamente não possua parâmetros, toda ferramenta já vem pré-configurada dessa forma, com valores padrão, você pode se sentir à vontade para ler a documentação de tal ferramenta e estudar a fundo o que cada parâmetro faz. Aqui ao longo dos exemplos serão dadas as devida explicações para os parâmetros que estão em uso, os internos devem ser consultados na documentação.

```

60 regl.predict([[6.2,3.4,5.4,2.3]])
61 regl.predict_proba([[6.2,3.4,5.4,2.3]])

```

Dando sequência, aplicando sob nossa variável regl as funções .predict() e .predict_proba(), passando como parâmetro uma linha escolhida aleatoriamente em nossa amostra, é realizada a devida previsão em forma de regressão logística.

```

In [ ]: regl.predict([[6.2,3.4,5.4,2.3]])
Out[ ]: array([2])

In [ ]: regl.predict_proba([[6.2,3.4,5.4,2.3]])
Out[ ]: array([[0.00167941, 0.1447824 , 0.85353819]])

```

Via console é possível ver o retorno da função .predict() que é array([2]), lembrando que nossa classificação é para valores entre 0 e 1 e 2, tal amostra analisada é do tipo 2 (virginica). Já o retorno de predict_proba() é interessante pois podemos ver que ele nos mostra uma array([[0.00, 0.14, 0.85]]) o que nos mostra que tal amostra tinha 0% de chance de ser do tipo 0, 14% de chance de ser do tipo 1 e 85% de chance de ser do tipo 2, o que é uma informação relevante do funcionamento do algoritmo.

```

62 previsor_regl = regl.predict(eteste)
63 margem_acertos_regl = metrics.accuracy_score(steste, previsor_regl)

```

Por fim criamos uma variável de nome previsor_regl que recebe como atributo o previsor de regl sob os valores de eteste. Por último criamos uma variável de nome margem_acertos_regl que aplica a função de taxa de precisão com base nos valores de steste e previsor_regl.

Nome	Tipo	Tamanho	Valor
k	int	1	25
margem_acertos	float64	1	0.9473684210526315
margem_acertos_regl	float64	1	1.0
previsor	int32	(38,)	[0 0 2 ... 2 0 1]
previsor_regl	int32	(38,)	[0 0 2 ... 2 0 1]
previsores_k	int32	(38,)	[0 0 2 ... 2 0 1]
rotulos	str320	(3,)	ndarray object of numpy module

Executando o último bloco de código podemos ver que são criadas as respectivas variáveis, uma com os valores previstos com base nos testes, assim como sua taxa de precisão (usando inclusive um dos valores de k que haviam dado como resultado 100% de precisão anteriormente), pois a taxa de previsão sobre as amostras se manteve em 100%.

Sendo assim terminamos o entendimento básico do processamento e análise de dados sobre a Iris Dataset usando modelo de rede neural simples.

Código Completo:

```
1 from sklearn.datasets import load_iris
2 from sklearn.neighbors import KNeighborsClassifier
3 from sklearn.model_selection import train_test_split
4 from sklearn import metrics
5 import matplotlib.pyplot as plt
6 from sklearn.linear_model import LogisticRegression
7
8 base = load_iris()
9 entradas = base.data
10 saidas = base.target
11 rotulos = base.target_names
12
13 knn = KNeighborsClassifier(n_neighbors = 1)
14 knn.fit(entradas, saidas)
15 knn.predict([[5.1,3.1,1.4,0.2]])
16
17 especie = knn.predict([[5.9,3.5,1.4,1.8]])[0]
18 print(especie)
19 rotulos[especie]
20
21 etreino, eteste, streino, steste = train_test_split(entradas,
22                                     saidas,
23                                     test_size = 0.25)
24 knn.fit(entreino, streino)
25 previsor = knn.predict(eteste)
26
27 margem_acertos = metrics.accuracy_score(steste, previsor)
28
29 valores_k = {}
30 k=1
31 while k < 25:
32     knn = KNeighborsClassifier(n_neighbors=k)
33     knn.fit(entreino, streino)
34     previsores_k = knn.predict(eteste)
35     acertos = metrics.accuracy_score(steste, previsores_k)
36     valores_k[k] = acertos
37     k += 1
38
39 plt.plot(list(valores_k.keys()),
40           list(valores_k.values()))
41 plt.xlabel('Valores de K')
42 plt.ylabel('Performance')
43 plt.show()
44
45 regl = LogisticRegression()
46 regl.fit(entreino, streino)
47 regl.predict([[6.2,3.4,5.4,2.3]])
48 regl.predict_proba([[6.2,3.4,5.4,2.3]])
49 previsor_regl = regl.predict(eteste)
50 margem_acertos_regl = metrics.accuracy_score(steste, previsor_regl)
```


Classificação Multiclasse via Rede Neural Artificial – Iris Dataset

Ao longo deste livro estaremos usando alguns datasets de exemplo disponibilizados em domínio público por meio de repositórios. Um dos maiores repositórios é o UCI Machine Learning Repository, mantido por algumas instituições educacionais e por doadores, conta atualmente com mais de 470 datasets organizados de acordo com o tipo de aprendizado de máquina a ser estudado e treinado.

The screenshot shows the homepage of the UC Irvine Machine Learning Repository. At the top, there's a logo featuring a kangaroo and the text "UCI Machine Learning Repository". Below the logo, it says "Center for Machine Learning and Intelligent Systems". On the right side, there are links for "About", "Citation Policy", "Donate a Data Set", and "Contact". There's also a search bar with options for "Repository" and "Web", and a "View ALL Data Sets" button. The main content area has sections for "Latest News", "Newest Data Sets", and "Most Popular Data Sets (hits since 2007)". The "Latest News" section lists recent updates. The "Newest Data Sets" section lists datasets added in 2019, with the first one being "Wave Energy Converters". The "Most Popular Data Sets" section lists datasets based on hits since 2007, with "Iris" at the top.

Most Popular Data Sets (hits since 2007):		
2748325:		Iris
1546708:		Adult
1199385:		Wine
1015217:		Car Evaluation
989304:		Wine Quality
976446:		Heart Disease

Na página inicial é possível ver que, da listagem de datasets mais populares o Iris ocupa a primeira posição, até mesmo em outras comunidades ele costuma ser o mais popular, sendo o passo inicial para a maioria dos cientistas de dados que iniciam seus estudos em aprendizado de máquina.

Machine Learning Repository
Center for Machine Learning and Intelligent Systems

Iris Data Set

Download: [Data Folder](#) [Data Set Description](#)

Abstract: Famous database; from Fisher, 1936



Data Set Characteristics:	Multivariate	Number of Instances:	150	Area:	Life
Attribute Characteristics:	Real	Number of Attributes:	4	Date Donated	1988-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	2748327

Source:

Creator:

R.A. Fisher

Abrindo a página dedicada ao Iris Dataset podemos ter uma descrição detalhada sobre do que se trata esse banco de dados assim como suas características. O que de imediato nos importa é que, como trabalhado anteriormente, trata-se de um banco de dados com 150 amostras das flores do tipo iris, divididas em 4 atributos previsores que usaremos para classificar tais amostras em 3 tipos de plantas diferentes. A única diferença por hora é que antes estávamos usando a base de dados exemplo contida na biblioteca SKLearn, agora iremos aprender como trabalhar com dados a partir de planilhas Excel, que será o mais comum eu seu dia-a-dia. Na própria página da Iris Dataset é possível fazer o download do banco de dados que iremos utilizar, uma vez feito o download do arquivo iris.csv podemos dar início ao código.

	A	B	C	D	E	F	G	H	
1	sepal length,sepal width,petal length,petal width,class								
2	5.1,3.5,1.4,0.2,Iris-setosa								
3	4.9,3.0,1.4,0.2,Iris-setosa								
4	4.7,3.2,1.3,0.2,Iris-setosa								
5	4.6,3.1,1.5,0.2,Iris-setosa								
6	5.0,3.6,1.4,0.2,Iris-setosa								
7	5.4,3.9,1.7,0.4,Iris-setosa								
8	4.6,3.4,1.4,0.3,Iris-setosa								
9	5.0,3.4,1.5,0.2,Iris-setosa								
10	4.4,2.9,1.4,0.2,Iris-setosa								
11	4.9,3.1,1.5,0.1,Iris-setosa								

Abrindo o arquivo iris.csv em nosso Excel é possível verificar como os dados estão organizados nessa base de dados.

Partindo para o Código:

```

1 import pandas as pd
2

```

Todo processo se inicia com a importação das bibliotecas e módulos que iremos utilizar ao longo de nosso código. Para este exemplo usaremos a biblioteca Pandas, o processo de importação segue os mesmos moldes de sempre, por meio do comando import assim como a referência pelo comando as como no exemplo acima.

```

3 base = pd.read_csv('iris.csv')
4

```

Em seguida criamos uma variável de nome base, por meio da função `read_csv()` do pandas conseguimos fazer a importação dos dados a partir de uma planilha Excel. Note que no parâmetro, em formato de string passamos o nome do arquivo assim como sua extensão, caso você tenha o arquivo com outro nome basta fazer a importação com o nome de seu arquivo. Outro ponto interessante é que nesse formato o interpretador busca o arquivo no mesmo local onde está salvo seu arquivo de código .py, caso você tenha salvo em uma pasta diferente é necessário colocar todo o caminho em forma de string como parâmetro.

Explorador de variáveis			
Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length petal width, cl ...

Se o processo de importação ocorreu da maneira certa no Explorador de Variáveis será possível ver a variável base assim como explorar seu conteúdo de forma visual.

base - DataFrame						
Index	sepal length	sepal width	petal length	petal width	class	
0	5.1	3.5	1.4	0.2	Iris-setosa	
1	4.9	3	1.4	0.2	Iris-setosa	
2	4.7	3.2	1.3	0.2	Iris-setosa	
3	4.6	3.1	1.5	0.2	Iris-setosa	
4	5	3.6	1.4	0.2	Iris-setosa	
5	5.4	3.9	1.7	0.4	Iris-setosa	
6	4.6	3.4	1.4	0.3	Iris-setosa	
7	2.4	1.5	0.2	0.2	Iris-setosa	

Formato Cor de fundo Min/max c

Repare que o objeto em questão é um DataFrame, este é um modelo parecido com o de matriz usado no exemplo anterior, onde tínhamos uma array do tipo Numpy, aqui, o sistema de

interpretador do Pandas transforma os dados em um DataFrame, a grosso modo a grande diferença é que um DataFrame possui uma modelo de indexação próprio que nos será útil posteriormente quando for necessário trabalhar com variáveis do tipo Dummy em exemplos mais avançados. Por hora, você pode continuar entendendo que é uma forma de apresentação dos dados com linhas e colunas, parecido com o modelo matricial.

```
5 entradas = base.iloc[:, 0:4].values
6 saídas = base.iloc[:, 4].values
```

Logo após criamos nossas variáveis que armazenarão os dados separados como características das amostras (entradas) assim como o tipo de planta (saídas). Para isso criamos nossa variável entradas que por meio da função .iloc[] e do comando .values pega todos valores contidos em todas as linhas e das 4 primeiras colunas de base (desconsiderando a coluna de índice). O mesmo é feito com nossa variável saídas, por meio da mesma função pega como atributo todos os valores de todas as linhas, mas somente da 4^a coluna (lembrando que as colunas com informação começam em 0, a 4^a coluna é a que contém os dados de saída).

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
saídas	object	(150,)	ndarray object of numpy module

Dessa forma separamos os dados de entrada e de saída de nossa base de dados, permitindo o cruzamento desses dados ou a aplicação de funções sobre os mesmos.

```
8 from sklearn.preprocessing import LabelEncoder
9
10 labelencoder = LabelEncoder()
```

Quando estamos fazendo nossos ajustes iniciais, na chamada fase de polimento dos dados, um dos processos mais comuns a se fazer é a conversão de nossos dados de base, todos para o mesmo tipo. Raciocine que é impossível fazer, por exemplo, uma multiplicação entre texto e número. Até o momento os dados que separamos estão em duas formas diferentes, os atributos de largura e comprimento das pétalas e sépalas em nossos dados de entrada são numéricos enquanto os dados de saída, referente a nomenclatura dos tipos de plantas classificadas, estão em formato de texto. Você até pode tentar fazer o cruzamento desses dados, mas terá um traceback mostrando a incompatibilidade entre os mesmos. Sendo assim, usaremos uma ferramenta capaz de converter texto em número para que possamos finalmente aplicar funções sobre os mesmos. A ferramenta em questão é a LabelEncoder, do módulo preprocessing da biblioteca sklearn. O processo de importação é feito como de praxe. Em seguida criamos uma variável labelencoder que inicializa essa ferramenta.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
saídas	object	(150,)	ndarray object of numpy module

Repare no explorador de variáveis que por hora o objeto saídas não é reconhecido corretamente pelo Numpy.

```

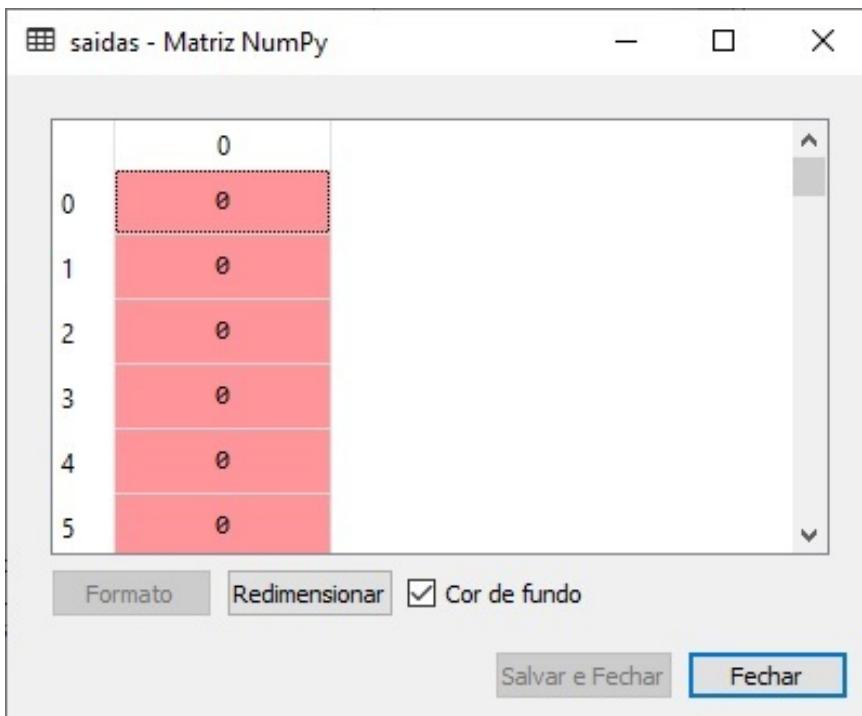
11 saídas = labelencoder.fit_transform(saidas)
12

```

Aplicando a função `.fit_transform()` de `labelencoder` sobre nossa variável `saídas` fazemos a devida conversão de dados do formato string para int.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, class [[5.1 3.5 1.4 0.2] [4.9 3.0 1.4 0.2]]
entradas	float64	(150, 4)	
saídas	int32	(150,)	[0 0 0 ... 2 2 2]

Executada a linha de código anterior, note que agora `saídas` possui seus dados em formato numérico, dentro de um intervalo entre 0 e 2.



Na sequência, outro processo que teremos de fazer, uma vez que processaremos nossos dados em uma rede neural densa, é a criação de variáveis do tipo Dummy. Variáveis Dummy são variáveis que criam uma indexação para nossa saída. Lembre-se que aqui estamos classificando 150 amostras em 3 tipos diferentes, porém os neurônios da camada de saída retornarão valores 0 ou 1. Dessa forma, precisamos criar um padrão de indexação que faça sentido tanto para o usuário quanto para o interpretador.

Em outras palavras, teremos de criar um padrão numérico binário para 3 saídas diferentes de acordo, nesse caso, com o tipo de planta. Por exemplo

Tipo de Planta	Neurônio 1	Neurônio 2	Neurônio 3
Setosa	0	0	1
Versicolor	0	1	0
Virginica	1	0	0

```

13 from keras.utils import np_utils
14
15 saidas_dummy = np_utils.to_categorical(saidas)

```

A nível de código, fazemos a importação da ferramenta np_utils, parte do módulo utils da biblioteca keras. Em seguida criamos uma variável de nome saidas_dummy que recebe sobre si como atributo a função np_utils.to_categorical() para que sejam criados os respectivos números de indexação como na tabela exemplo acima, para cada uma das amostras da base de dados.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ... [[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
entradas	float64	(150, 4)	
saidas	int32	(150,)	[0 0 0 ... 2 2 2]
saidas_dummy	float32	(150, 3)	[[1. 0. 0.] [1. 0. 0.]]

Selecionando e executando mais esse bloco de código podemos ver que foi criada a variável saidas_dummy, abrindo-a você verá o padrão como o da tabela, para cada uma das 150 amostras.

```

17 from sklearn.model_selection import train_test_split
18
19 etreino, eteste, streino, steste = train_test_split(entradas,
20                                     saidas_dummy,
21                                     test_size = 0.25)

```

Dando sequência, como havíamos feito no exemplo anterior, aqui novamente podemos dividir nossa base de dados em partes e realizar testes com essas partes para avaliação da performance do modelo. Também é possível aplicar o processamento da rede diretamente sobre nossa base de dados, porém uma prática recomendável para se certificar da integridade dos dados é os processar em partes distintas e realizar a comparação.

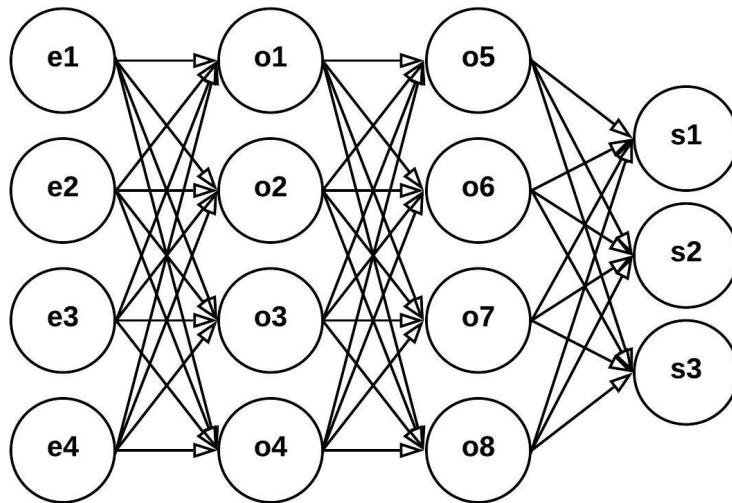
Dessa forma, criamos quatro variáveis dedicadas a partes para treino e teste de nossa rede neural. Logo etreino, eteste, streino, steste recebem como atributo train_test_split, que por sua vez recebe como parâmetro os dados contidos em entradas, saidas_dummy e por fim é definido que 25% das amostras sejam separadas para teste, enquanto os outros 75% serão separadas para treino.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ... [[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
eteste	float64	(38, 4)	[[5.8 2.7 4.1 1.] [6.7 3.3 5.7 2.5]]
etreino	float64	(112, 4)	[[6.7 2.5 5.8 1.8] [4.9 3. 1.4 0.2]]
saidas	int32	(150,)	[0 0 0 ... 2 2 2]
saidas_dummy	float32	(150, 3)	[[1. 0. 0.] [1. 0. 0.]]
steste	float32	(38, 3)	[[0. 1. 0.] [0. 0. 1.]]

Se todo processo correu normalmente é possível visualizar no explorador de variáveis tais objetos.

```
23 from keras.models import Sequential  
24 from keras.layers import Dense
```

Finalmente vamos dar início a criação da estrutura de nossa rede neural densa, como sempre, inicialmente é necessário fazer as devidas importações para que possamos usufruir das ferramentas adequadas. Então do módulo models da biblioteca keras importamos Sequential, da mesma forma, do módulo layers importamos Dense. Sequential é um modelo pré-configurado onde conseguimos criar e fazer as devidas conexões entre camadas de neurônios, Dense por sua vez permite a conexão entre todos nós da rede assim como a aplicação de funções em todas etapas da mesma.



Numa representação visual como fizemos anteriormente com os perceptrons, uma rede densa como a que estaremos criando agora seguiria uma estrutura lógica como a da figura acima.

```
26 classificador = Sequential()  
27 classificador.add(Dense(units = 4,  
28                      activation = 'relu',  
29                      input_dim = 4))
```

Inicialmente criamos uma variável de nome classificador, que recebe como atributo a inicialização da ferramenta Sequential, aqui, por enquanto, sem parâmetros mesmo. Em seguida por meio da função .add() começamos de fato a criação da camada de entrada e da sua comunicação com a primeira camada oculta. Para isso passamos como parâmetro para add Dense que por sua vez tem como parâmetros units = 4, ou seja, 4 neurônios na primeira camada oculta, activation = 'relu' (Rectified Linear Units, em tradução livre Unidades Linearmente Retificadas, algo como a StepFunction usada no exemplo anterior, porém entre camadas de redes neurais quando aplicamos uma função degrau é por meio da relu), que é o método de ativação dessa camada, e por fim input_dim = 4, que é o parâmetro que determina quantos neurônios existem em nossa camada de entrada, aqui, 4 tipos de características das plantas representam 4 neurônios da camada de entrada.

```
30 classificador.add(Dense(units = 4,  
31                      activation = 'relu'))
```

Logo após criamos mais uma camada oculta, também com 4 neurônios e também recebendo como função de ativação relu. Repare que aqui não é mais necessário especificar os neurônios de entrada, raciocine que esta camada é a conexão entre a primeira camada oculta e a camada de saída dessa rede neural.

```
32 classificador.add(Dense(units = 3,
33                         activation = 'softmax'))
```

Dando sequência criamos a camada de saída dessa rede, nos mesmos moldes das anteriores, porém com a particularidade que na camada de saída temos 3 neurônios, já que estamos classificando as amostras em 3 tipos, e o método de ativação agora é o ‘softmax’, método esse que segundo a documentação do keras é o que mais se aproxima da função sigmoide que usamos anteriormente, a diferença basicamente é que sigmoid oferece a probabilidade de ser de um tipo ou de outro (2 tipos), enquanto softmax suporta mostrar a probabilidade de uma amostra ser classificada em 3 ou mais tipos.

```
34 classificador.compile(optimizer = 'adam',
35                       loss = 'categorical_crossentropy',
36                       metrics = ['categorical_accuracy'])
```

Prosseguindo, por meio da função .compile() criamos o compilador para nossa rede neural, nesta etapa é necessário passar três parâmetros, o primeiro deles, optimizer = ‘adam’, com o nome autoexplicativo, aplica alguns métodos para otimização dos resultados dos processamentos, loss = ‘categorical_crossentropy’ é a nossa função de perda, raciocine que em uma rede neural há uma pequena parcela das amostras que podem ter dados inconsistentes ou erros de processamento, estas caem nessa categoria, o parâmetro da função de perda por sua vez tentará pegar essas amostras fora da curva e tentar encaixar de acordo com a semelhança/proximidade, a outras amostras que foram processadas corretamente, por fim metrics = [‘categorical_accuracy’] faz a avaliação interna do modelo, mensurando sua precisão quanto ao processo de categorizar corretamente as amostras.

```
37 classificador.fit(entreino,
38                     streino,
39                     batch_size = 10,
40                     epochs = 1000)
```

Por fim, por meio da função .fit() faremos a alimentação de nossa rede e definiremos os parâmetros de processamento dela. Sendo assim, .fit() recebe como parâmetros os dados de entreino e de streino, batch_size = 10 diz respeito a taxa de atualização dos pesos, ou seja, de quantas em quantas amostras serão testadas, corrigidas e testadas e por fim epochs = 1000 define quantas vezes a rede neural será executada. Esses parâmetros na prática devem inclusive serem modificados e testados para tentar sempre encontrar a melhor performance do algoritmo. Em certos casos a taxa de atualização pode influenciar bastante a diminuição da margem de erro, assim como executar a rede neural mais vezes faz o que em algumas literaturas é chamado de aprendizagem por reforço, toda rede neural começa com rápidas atualizações e correções, porém ela chega em níveis onde após a execução de algumas milhares (ou até mesmo milhões) de vezes executada ela pode acabar estagnada em um valor.

```

112/112 [=====] - 0s 125us/step - loss: 0.0585 -
categorical_accuracy: 0.9821
Epoch 996/1000
112/112 [=====] - 0s 125us/step - loss: 0.0584 -
categorical_accuracy: 0.9732
Epoch 997/1000
112/112 [=====] - 0s 125us/step - loss: 0.0584 -
categorical_accuracy: 0.9821
Epoch 998/1000
112/112 [=====] - 0s 125us/step - loss: 0.0552 -
categorical_accuracy: 0.9821
Epoch 999/1000
112/112 [=====] - 0s 125us/step - loss: 0.0551 -
categorical_accuracy: 0.9821
Epoch 1000/1000
112/112 [=====] - 0s 143us/step - loss: 0.0562 -
categorical_accuracy: 0.9821
Out[ ]:

```

Selecionando todo bloco de código da estrutura da rede neural e executando, você pode acompanhar pelo terminal a mesma sendo executada, assim como acompanhar a atualização da mesma. Aqui, neste exemplo, após a execução da mesma 1000 vezes para o processo de classificação, categorical_accuracy: 0.98 nos mostra que a taxa de precisão, a taxa de acertos na classificação das 150 amostras foi de 98%.

Tenha em mente que bases de dados grandes ou redes que são executadas milhões de vezes, de acordo com seu hardware essa rede pode levar um tempo considerável de processamento. Aqui nesse exemplo, 150 amostras, 3 categorias, 1000 vezes processadas, toda tarefa é feita em poucos segundos, mas é interessante ter esse discernimento de que grandes volumes de dados demandam grande tempo de processamento.

Terminada a fase de processamento da rede neural, hora de fazer algumas avaliações de sua performance para conferir e confirmar a integridade dos resultados.

```

42 avalPerformance = classificador.evaluate(eteste, steste)
43
44 previsoes = classificador.predict(eteste)
45 previsoesVF = (previsoes > 0.5)

```

Para isso inicialmente criamos uma variável de nome avalPerformance que recebe como atributo a função .evaluate() de nosso classificador, tendo como parâmetros os dados de eteste e steste. Também criamos uma variável de nome previsoes, que recebe a função .predict() sobre os dados de eteste. Por fim, criamos uma variável de nome previsoesVF que basicamente retorna os resultados de previsões em formato True ou False, por meio da condicional previsoes > 0.5.

Nome	Tipo	Tamanho	Valor
avalPerformance	list	2	[0.03765642162608473, 0.9736842105263158]
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, cl ...
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
eteste	float64	(38, 4)	[[5.8 2.7 4.1 1.] [6.7 3.3 5.7 2.5]]
etreino	float64	(112, 4)	[[6.7 2.5 5.8 1.8] [4.9 3. 1.4 0.2]]
previsoes	float32	(38, 3)	[[3.6104400e-05 9.9992716e-01 3.6696962e-01 [5.5724660e-18 7.9111029 ...
previsoesVF	bool	(38, 3)	[[False True False] [False False True]]

Selecionando e executando esse bloco de código é possível ver a criação das devidas variáveis.

A screenshot of a Jupyter Notebook cell titled "avalPerformance - Lista (2 elementos)". The cell displays a table with two rows:

Índice	Tipo	Tamanho	Valor
0	float64	1	0.03765642162608473
1	float64	1	0.9736842105263158

Buttons at the bottom: "Salvar e Fechar" (Save and Close) and "Fechar" (Close).

Abrindo `avalPerformance` temos dados para comparar com os do processamento da rede neural. A rede neural havia nos retornado uma taxa de precisão de 98%, enquanto aqui realizado testes de outras formas pela função `.evaluate()` o retorno foi de 97%. O retorno de loss function da rede havia sido de 0.05%, o da `evaluate` 0.03%. O que confirma uma excelente precisão nos resultados dos processamentos, tenha em mente que essa fase de avaliação é dada até mesmo como opcional em algumas literaturas, porém para uma atividade profissional é imprescindível que você se certifique ao máximo da integridade de seus resultados.

A screenshot of a Jupyter Notebook cell titled "previsões - Matriz NumPy". The cell displays a table with 6 rows and 3 columns:

	0	1	2
0	3.61044e-05	0.999927	3.6697e-05
1	5.57247e-18	7.9111e-05	0.999921
2	2.45434e-06	0.999437	0.00056071
3	5.47915e-08	0.992885	0.00711488
4	2.62146e-08	0.93504	0.0649605
5	0.000404379	0.999546	4.93405e-05

Buttons at the bottom: "Formato" (Format), "Redimensionar" (Resize), "Cor de fundo" (Background color), "Salvar e Fechar" (Save and Close), and "Fechar" (Close).

Abrindo `previsões` podemos ver a probabilidade de cada amostra ser de um determinado tipo, por exemplo, na linha 0, em azul, há o dado probabilístico de 99% de chance dessa amostra ser do tipo 1 (versicolor), na segunda linha, 99% de chance de tal amostra ser do tipo 2 (virginica) e assim por diante.

previsoesVF - Matriz NumPy

	0	1	2	
0	False	True	False	^
1	False	False	True	
2	False	True	False	
3	False	True	False	
4	False	True	False	
5	False	True	False	▼

Formato Redimensionar Cor de fundo

Salvar e Fechar Fechar

Abrindo previsoesVF podemos ver de forma ainda mais clara que a amostra da linha 0 é do tipo 1, que a amostra da linha 1 é do tipo 2, e assim subsequente para todas as amostras.

Outra metodologia de avaliação de nosso modelo é feita pela chamada Matriz de Confusão, método que nos mostrará quantas classificações foram feitas corretamente e as que foram classificadas erradas para cada saída. Para aplicar a matriz de confusão em nosso modelo precisaremos fazer algumas alterações em nossos dados, assim como fizemos lá no início na fase de polimento.

```
47 import numpy as np
48
49 steste2 = [np.argmax(t) for t in steste]
50 previsoes2 = [np.argmax(t) for t in previsoes]
```

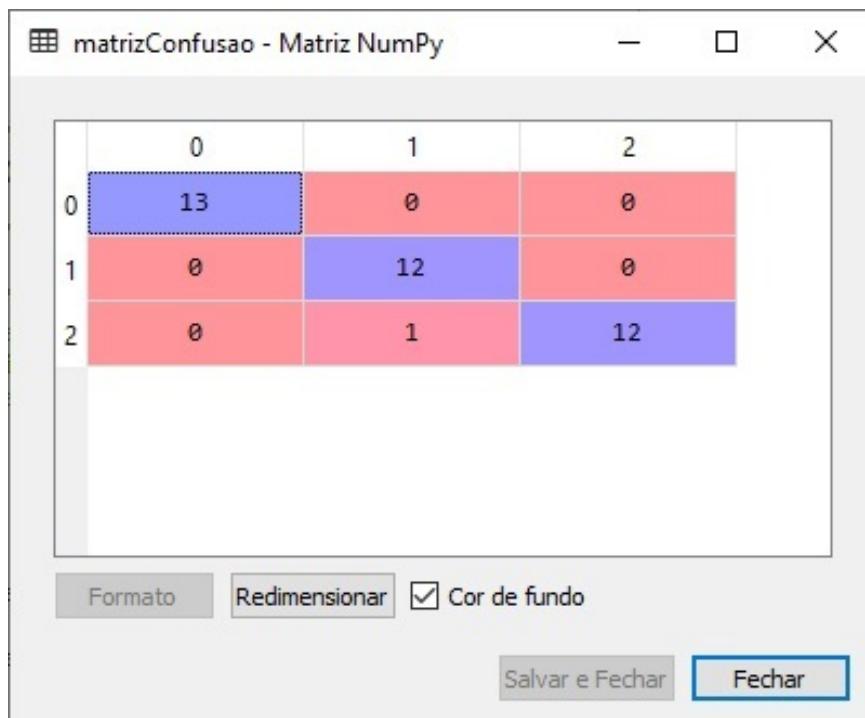
Dessa vez importamos a biblioteca Numpy e a referenciamos como np. Em seguida criamos uma variável de nome steste2 que recebe como atributo a função np.argmax(t), que pega os valores de steste e gera uma indexação própria, contendo a posição da matriz onde a planta foi classificada. O mesmo é feito em previsões e atribuído para previsões2.

```
52 from sklearn.metrics import confusion_matrix
53
```

Logo a seguir fazemos mais uma importação necessária, dessa vez, da ferramenta confusion_matrix do módulo metrics da biblioteca sklearn.

```
54 matrizConfusao = confusion_matrix(previsoes2, steste2)
55
```

Dando sequência criamos uma variável de nome matrizConfusao que por sua vez realiza a aplicação dessa ferramenta sobre previsões2 e steste2.



Selecionando e executando o bloco de código acima é possível ver que foi criada a variável matrizConfusao, explorando seu conteúdo podemos fazer a leitura, em forma de verificação que: Para o tipo de planta 0, foram classificadas 13 amostras corretamente, para o tipo de planta 1, foram classificadas 12 amostras corretamente, e para o tipo de planta 2, foram classificadas 12 amostras de maneira correta e 1 amostra de maneira incorreta.

Finalizando este modelo, outra prática muito usada que é interessante já darmos início a seu entendimento é a chamada Validação Cruzada. Este método é de suma importância para se certificar de que seus testes não estão “viciados”, uma vez que uma rede neural mal configurada pode entrar em loops de repetição ou até mesmo retornar resultados absurdos. A técnica em si consiste em pegar nossa base de dados inicial, separar a mesma em partes iguais e aplicar o processamento da rede neural individualmente para cada uma das partes, para por fim se certificar que o padrão de resultados está realmente correto.

```
56 from sklearn.model_selection import cross_val_score
57
```

Dessa vez importamos a ferramenta cross_val_score do módulo model_selection da biblioteca sklearn.

```
58 def valCruzada():
59     classificadorValCruzada = Sequential()
60     classificadorValCruzada.add(Dense(units = 4,
61                                         activation = 'relu',
62                                         input_dim = 4))
63     classificadorValCruzada.add(Dense(units = 4,
64                                         activation = 'relu'))
65     classificadorValCruzada.add(Dense(units = 3,
66                                         activation = 'softmax'))
67     classificadorValCruzada.compile(optimizer = 'adam',
68                                     loss = 'categorical_crossentropy',
69                                     metrics = ['categorical_accuracy'])
70
71 return classificadorValCruzada
```

Em seguida criamos uma rede interna em forma de função chamada ValCruzada, sem parâmetros mesmo. Repare que sua estrutura interna é a mesma da rede criada anteriormente com a exceção que agora não a alimentaremos por meio da função `.fit()` mas sim fazendo o uso do `KerasClassifier`.

```
72 from keras.wrappers.scikit_learn import KerasClassifier  
73
```

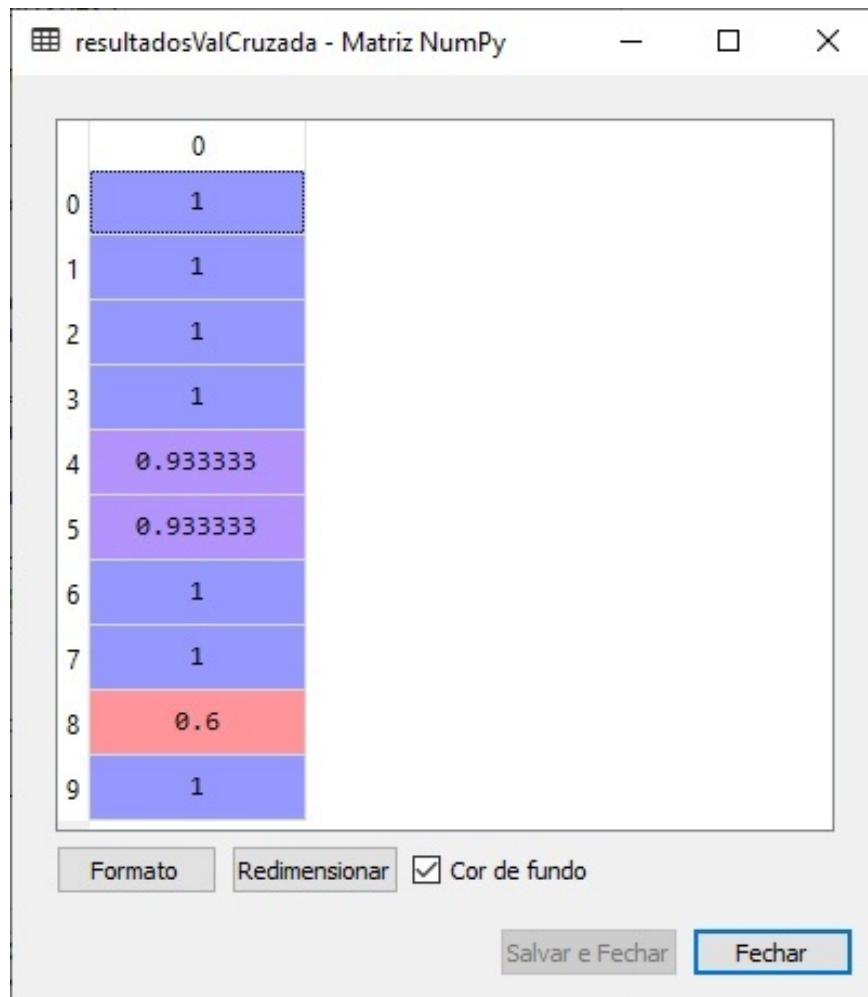
Como o `KerasClassifier` ainda não havia sido incorporado em nosso código é necessário fazer a importação do mesmo, nos mesmos moldes de sempre.

```
74 classificadorValCruzada = KerasClassifier(build_fn = valCruzada,  
75                                         epochs = 1000,  
76                                         batch_size = 10)
```

Criamos uma variável a executar a ferramenta `KerasClassifier` de nome `classificadorValCruzada`. Tal ferramenta por sua vez recebe como parâmetros `build_fn = valCruzada`, o que significa que a função de ativação desse classificador é a própria rede neural criada anteriormente de nome `valCruzada`, outro parâmetro é `epochs = 1000`, ou seja, definindo que o processamento da rede será feito mil vezes e por fim `batch_size = 10`, parâmetro que defini de quantos em quantos pesos seus valores serão corrigidos.

```
78 resultadosValCruzada = cross_val_score(estimator = classificadorValCruzada,  
79                                         X = entradas,  
80                                         y = saídas,  
81                                         cv = 10,  
82                                         scoring = 'accuracy')
```

Na sequência criamos uma variável chamada `resultadosValCruzada` que executará a própria ferramenta `cross_val_score`, que por sua vez recebe como parâmetros `estimator = classificadorValCruzada`, o que em outras palavras nada mais é do que aplicar essa função sobre o processamento da rede `valCruzada`, `X = entradas` e `y = saídas`, auto sugestivo, variáveis temporárias com os dados contidos em entradas e saídas, `cv = 10`, parâmetro que define em quantas partes nossa base de dados será dividida igualmente para ser processada pela rede e por fim `scoring = 'accuracy'` define o método a ser formatados os resultados, aqui, com base em precisão de resultados.



Abrindo a variável resultadosValCruzada é possível ver de forma bruta, mas fácil conversão para percentual, os resultados do processamento da rede sobre cada amostra. Note que aqui houve inclusive uma exceção onde o processamento em cima da amostra nº 8 teve uma margem de acerto de apenas 60%, enquanto para todas as outras a margem ficou dentro de um intervalo entre 93% e 100%. Isto pode acontecer por diversos fatores, os mais comuns, erros de leitura dos dados da base ou aplicação de parâmetros errados sobre a rede neural.

```

84 media = resultadosValCruzada.mean()
85 desvio = resultadosValCruzada.std()

```

Por fim, apenas para apresentar nossos dados de forma mais clara, criamos uma variável de nome media que recebe o valor médio entre todos valores obtidos em resultadosValCruzada, assim como é interessante apresentar o cálculo do desvio (algo como a margem de erro em percentual), aqui inclusive, este valor vai sofrer um grande impacto devido ao resultado da validação cruzada sobre a 8^a amostra.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(150, 5)	Column names: sepal length, sepal width, petal length, petal width, class
desvio	float64	1	0.11850925889754119
entradas	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]]
eteste	float64	(38, 4)	[[6.8 3.2 5.9 2.3] [5.1 3.8 1.6 0.2]]
etreino	float64	(112, 4)	[[6.1 3. 4.9 1.8] [6.5 3. 5.2 2.]]
matrizConfusao	int64	(3, 3)	[[0 0 0] [13 10 15]]
media	float64	1	0.9466666666666667

Via explorador de variáveis podemos ver facilmente os valores de media (taxa de acerto de 94%) e de desvio (11% de margem de erro para mais ou para menos, o que é um número absurdo, porém justificável devido ao erro de leitura da 8ª amostra anterior).

Código Completo:

```

1 import pandas as pd
2 import numpy as np
3 from sklearn.preprocessing import LabelEncoder
4 from keras.utils import np_utils
5 from sklearn.model_selection import train_test_split
6 from keras.models import Sequential
7 from keras.layers import Dense
8 from sklearn.metrics import confusion_matrix
9 from sklearn.model_selection import cross_val_score
10 from keras.wrappers.scikit_learn import KerasClassifier
11
12 base = pd.read_csv('iris.csv')
13 entradas = base.iloc[:, 0:4].values
14 saidas = base.iloc[:, 4].values
15
16 labelencoder = LabelEncoder()
17 saidas = labelencoder.fit_transform(saidas)
18 saidas_dummy = np_utils.to_categorical(saidas)
19
20 etreino, eteste, streino, steste = train_test_split(entradas,
21                                     saidas_dummy,
22                                     test_size = 0.25)
23 classificador = Sequential()
24 classificador.add(Dense(units = 4,
25                         activation = 'relu',
26                         input_dim = 4))
27 classificador.add(Dense(units = 4,
28                         activation = 'relu'))
29 classificador.add(Dense(units = 3,
30                         activation = 'softmax'))
31 classificador.compile(optimizer = 'adam',
32                       loss = 'categorical_crossentropy',
33                       metrics = ['categorical_accuracy'])
34 classificador.fit(etreino,
35                   streino,
36                   batch_size = 10,
37                   epochs = 1000)

```

```
39 avalPerformance = classificador.evaluate(eteste, steste)
40 previsoes = classificador.predict(eteste)
41 previsoesVF = (previsoes > 0.5)
42 steste2 = [np.argmax(t) for t in steste]
43 previsoes2 = [np.argmax(t) for t in previsoes]
44 matrizConfusao = confusion_matrix(previsoes2, steste2)
45
46 def valCruzada():
47     classificadorValCruzada = Sequential()
48     classificadorValCruzada.add(Dense(units = 4,
49                                     activation = 'relu',
50                                     input_dim = 4))
51     classificadorValCruzada.add(Dense(units = 4,
52                                     activation = 'relu'))
53     classificadorValCruzada.add(Dense(units = 3,
54                                     activation = 'softmax'))
55     classificadorValCruzada.compile(optimizer = 'adam',
56                                     loss = 'categorical_crossentropy',
57                                     metrics = ['categorical_accuracy'])
58     return classificadorValCruzada
59
60 classificadorValCruzada = KerasClassifier(build_fn = valCruzada,
61                                            epochs = 1000,
62                                            batch_size = 10)
63
64 resultadosValCruzada = cross_val_score(estimator = classificadorValCruzada,
65                                         X = entradas,
66                                         y = saidas,
67                                         cv = 10,
68                                         scoring = 'accuracy')
69 media = resultadosValCruzada.mean()
70 desvio = resultadosValCruzada.std()
```

Classificação Binária via Rede Neural Artificial – Breast Cancer Dataset

Uma das aplicações mais comuns em data science, como você deve ter percebido, é a classificação de dados para os mais diversos fins. Independentemente do tipo de dados e da quantidade de dados a serem processados, normalmente haverá uma fase inicial onde será feito o tratamento desses dados para que se possa extrair informações relevantes dos mesmos.

Tenha em mente que por vezes o problema que estaremos abstraindo é a simples classificação de dados, porém haverão situações onde a classificação será apenas parte do tratamento dos mesmos. Se tratando de redes neurais é comum fazer o seu uso para identificação de padrões a partir de dados alfanuméricos ou imagens e com base nesses padrões realizar aprendizado de máquina para que seja feita a análise de dados.

Segundo esse raciocínio lógico, uma das aplicações que ganhou destaque e vem evoluindo exponencialmente é a aplicação de redes neurais dentro da imaginologia médica, exames de imagem computadorizados como raios-x, tomografia computadorizada, ressonância magnética hoje possuem tecnologia para se obter imagens de alta definição fidedignas da anatomia humana para que se possam investigar possíveis patologias.

Ainda dentro da radiologia médica, uma das mais importantes aplicações de redes neurais foi para contribuir com a identificação de diferentes tipos de câncer a partir de exames de imagem. Tenha em mente que a ideia não é jamais substituir o profissional médico radiologista que é o especialista nisso, mas oferecer ferramentas que auxiliem ou de alguma forma aumente a precisão de seus diagnósticos para que os pacientes recebam tratamentos mais adequados.

Como exemplo deste tipo de aplicação de rede neural, usaremos um dataset bastante conhecido que com base num banco de dados de mais de 560 imagens de exames de mamografia para treinar uma rede neural artificial que possa a partir dessa estrutura, classificar imagens quanto a presença ou não de tumores. Por fim esse modelo pode ser adaptado para a partir de novas imagens realizar tal classificação, ou salvo para novos problemas computacionais que de alguma forma possam envolver classificação binária, classificação do tipo “ou uma coisa ou outra”.

Lembrando que, como mencionado algumas vezes nos capítulos anteriores, estaremos trabalhando de forma procedural, cada vez será usado um referencial teórico mais enxuto enquanto avançaremos para exemplos práticos de maior complexidade.

```
1 import pandas as pd
2
3 entradas = pd.read_csv('entradas-breast.csv')
4 saidas = pd.read_csv('saídas-breast.csv')
```

Como sempre o processo se inicia com a importação das bibliotecas, módulos e ferramentas a serem utilizados. De imediato importamos a biblioteca Pandas e a referenciamos como pd, para que na sequência possamos por meio de sua função `.read_csv()` importar dados a partir de planilhas Excel.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ...
saidas	DataFrame	(569, 1)	Column names: 0

Se o processo de importação correu como esperado no explorador de variáveis é possível visualizar tais variáveis. Como estamos trabalhando inicialmente com a biblioteca pandas note que os dados são importados e reconhecidos como DataFrame, em outras bibliotecas essa nomenclatura é a usual matriz. Vale lembrar que para importação acontecer corretamente o arquivo deve estar na mesma pasta onde você salvou seu arquivo de código, ou especificar o caminho completo no parâmetro de importação.

Index	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
0	17.99	10.38	122.8	1001	0.1184	0.2776
1	20.57	17.77	132.9	1326	0.08474	0.07864
2	19.69	21.25	130	1203	0.1096	0.1599
3	11.42	20.38	77.58	386.1	0.1425	0.2839
4	20.29	14.34	135.1	1297	0.1003	0.1328
5	12.45	15.7	82.57	477.1	0.1278	0.17
6	18.25	19.98	119.6	1040	0.09463	0.109
7	13.71	20.83	90.2	577.9	0.1189	0.1645
8	13	21.82	87.5	519.8	0.1273	0.1932
9	12.46	24.04	83.97	475.9	0.1186	0.2396
10	16.02	23.24	102.7	797.8	0.08206	0.06669
11	15.78	17.89	103.6	781	0.0971	0.1292
12	19.17	24.8	132.4	1123	0.0974	0.2458
13	15.85	23.95	103.7	782.7	0.08401	0.1002

Abrindo entradas podemos ver que de fato, existem 569 amostras, com 30 parâmetros para cada uma delas (valores individuais, de perda e de média para raio, textura, perímetro, área, borramento, compactação, concavidade, pontos de cavidade, simetria, dimensão fractal). Com base em todos esses dados as amostras serão classificadas como positivo ou negativo para o diagnóstico de câncer.

```
6 from sklearn.model_selection import train_test_split
7
8 etreino, eteste, streino, steste = train_test_split(entradas,
9                                     saidas,
10                                    test_size = 0.25)
```

Em seguida realizamos a importação da ferramenta train_test_split e dividimos nossas amostras de nossa base de dados para que possam ser treinadas e testadas individualmente por nossa rede

neural. Lembrando que esta etapa é uma fase de certificação da integridade de nossos processos e resultados, valores próximos em comparação mostram que nossa rede está configurada corretamente.

```
12 import keras
13 from keras.models import Sequential
14 from keras.layers import Dense, Dropout
```

Em seguida fazemos mais umas importações, primeiro, da biblioteca keras, depois dos módulos Sequential e Dense que como já havíamos feito outrora, servem como estruturas pré-configuradas de nossa rede multicamada interconectada. O único diferencial aqui é que do módulo layers da biblioteca keras também importamos Dropout. Esse módulo usaremos na sequência, por hora, raciocine que quando estamos trabalhando com redes neurais com muitas camadas, ou muitos neurônios por camada, onde esses tem valores de dados muito próximos, podemos determinar que aleatoriamente se pegue um percentual desses neurônios e simplesmente os subtraia da configuração para ganharmos performance. Em outras palavras, imagine uma camada com 30 neurônios, onde cada neurônio desse faz suas 30 conexões com a camada subsequente, sendo que o impacto individual ou de pequenas amostras deles é praticamente nulo na rede, sendo assim, podemos reduzir a quantia de neurônios e seus respectivos nós ganhando performance em processamento e não perdendo consistência dos dados.

```
16 classificador = Sequential()
17 classificador.add(Dense(units = 16,
18                         activation = 'relu',
19                         kernel_initializer = 'random_uniform',
20                         input_dim = 30))
21 classificador.add(Dense(units = 1,
22                         activation = 'sigmoid'))
23 classificador.compile(optimizer = 'adam',
24                         loss = 'binary_crossentropy',
25                         metrics = ['binary_accuracy'])
26 classificador.fit(streino,
27                     streino,
28                     batch_size = 10,
29                     epochs = 100)
```

Logo após criamos aquela estrutura de rede neural que estamos começando a ficar familiarizados. Repare nas diferenças em relação ao exemplo anterior, desta vez, aqueles 30 parâmetros de classificação de cada amostra é transformado em neurônios de nossa camada de entrada, a camada oculta por sua vez tem 16 neurônios e a camada de saída apenas um, pois trata-se de uma classificação binária (0 ou 1, 0 para negativo e 1 para positivo de câncer para uma devida amostra). Outra grande diferença que agora nessa estrutura, na camada onde há a fase de compilação as funções de perda e métricas também são diferentes do modelo anterior ,agora não estamos classificando objetos de forma categórica, mas de forma binária. Por fim, lembre-se que uma vez definida a fase onde existe a função .fit(), ao selecionar e executar esse bloco de código a rede começará seu processamento, aqui, alimentada com os dados contidos em streino e streino, atualizando os valores a cada 10 rodadas por meio do parâmetro batch_size = 10 e executando o reprocessamento da rede 100 vezes de acordo com o parâmetro epochs = 100.

```

Epoch 96/100
426/426 [=====] - 0s 106us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 97/100
426/426 [=====] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 98/100
426/426 [=====] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 99/100
426/426 [=====] - 0s 110us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Epoch 100/100
426/426 [=====] - 0s 108us/step - loss: 5.9503 -
binary_accuracy: 0.6268
Out[ ]:

```

Executada uma primeira vez a rede com esses parâmetros note que ela nos retorna uma taxa de precisão sobre sua classificação binária de apenas 62%, o que pode ser interpretada como uma margem de erro muito alta nesses moldes de processamento, o que precisamos fazer nesses casos são reajustes para aumentar essa taxa de precisão/acertos, em problemas reais, para fins profissionais, dependendo do contexto uma taxa de acerto por volta de 90% já é considerada baixa... 62% está muito fora desse padrão.

```

31 previsor = classificador.predict(eteste)
32

```

Criamos nosso previsor, que realiza as devidas previsões sobre eteste por meio da função .predict().

```

33 from sklearn.metrics import confusion_matrix, accuracy_score
34
35 margem_acertos = accuracy_score(steste, previsor)
36 matriz_confusao = confusion_matrix(steste, previsor)
37 resultadoMatrizConfusao = classificador.evaluate(eteste, steste)

```

Realizando as importações das ferramentas confusion_matrix e accuracy_score do módulo metrics da biblioteca sklearn podemos realizar estes testes complementar e verificar se existem erros ou inconsistências no processamento desses dados ou se realmente por hora a eficiência de nossa rede está muito baixa. Criamos então uma variável de nome margem_acertos que aplica a função da ferramenta accuracy_score sobre os dados de steste e previsor. Da mesma forma criamos uma variável de nome matriz_confusao que aplica o teste homônimo também sobre os dados de steste e previsor. Por fim criamos uma variável de nome resultadoMatrizConfusão que faz o levantamento desse parâmetro de comparação sobre os dados de eteste e steste.

Nome	Tipo	Tamanho	Valor
eteste	DataFrame	(143, 30)	Column names: radius_mean, text
etreino	DataFrame	(426, 30)	Column names: radius_mean, text
margem_acertos	float64	1	0.6293706293706294
matriz_confusao	int64	(2, 2)	[[0 53] [0 90]]
previsor	float32	(143, 1)	[[1.] [1.]]
resultadoMatrizConfusao	list	2	[5.908716241796534, 0.6293706295790372]
saidas	DataFrame	(569, 1)	Column names: 0

Selecionando todo bloco de código e executando é possível verificar nas variáveis criadas que de fato se comprovou que, executando testes individuais, o padrão de 62% de taxa de acerto se manteve. Identificado que o problema é a eficiência de nossa rede neural, hora de trabalhar para corrigir isso. Mas antes, como de praxe, vamos executar a validação cruzada sobre nossa base de dados.

```
39 from keras.wrappers.scikit_learn import KerasClassifier  
40 from sklearn.model_selection import cross_val_score
```

Para isso, vamos aproveitar e já fazer as importações do KerasClassifier assim como do cross_val_score, das suas respectivas bibliotecas como mostrado acima.

```
42 def valCruzada():  
43     classificadorValCruzada = Sequential()  
44     classificadorValCruzada.add(Dense(units = 16,  
45                             activation = 'relu',  
46                             kernel_initializer = 'random_uniform',  
47                             input_dim = 30))  
48     classificador.add(Dropout(0.2))  
49     classificadorValCruzada.add(Dense(units = 16,  
50                             activation = 'relu',  
51                             kernel_initializer = 'random_uniform'))  
52     classificadorValCruzada.add(Dense(units = 1,  
53                             activation = 'sigmoid'))  
54     classificadorValCruzada.compile(optimizer = 'adam',  
55                             loss = 'binary_crossentropy',  
56                             metrics = ['binary_accuracy'])  
57  
58 return classificadorValCruzada
```

Dando sequência, assim como feito no exemplo anterior, vamos criar nossa validação cruzada em forma de função. Para isso definimos valCruzada, sem parâmetros mesmo, e dentro dela criamos aquela estrutura de rede neural densa e sequencial como estamos habituados. Note que o único diferencial é que entre a primeira camada oculta e a cama oculta subsequente existe uma cama de Dropout, parametrizada em 0.2, na prática o que esta camada faz é, do seu número total de neurônios da camada, pegar 20% deles aleatoriamente e simplesmente os descartar de forma a não gerar impacto negativo no processamento. Inicialmente essa camada que, havia 16 neurônios, passa a ter 13, e dessa forma, somando os outros métodos de reajustes dos pesos e processamento entre camadas, se busca uma integridade de dados igual, porém com performance maior. Raciocine que eu uma camada de 16 neurônios este impacto é mínimo, mas em uma rede que, apenas como exemplo, possua mais de 100 neurônios por camada, assim como mais de 5 camadas ocultas, o parâmetro definido como dropout para elas irá gerar grande impacto de performance. Como sempre, é interessante realizar testes com diferentes valores de parâmetro e comparar os resultados.

```
59 classificador = KerasClassifier(build_fn = valCruzada,  
60                                 epochs = 100,  
61                                 batch_size = 10)  
62 resultadoValCruzada = cross_val_score(estimator = classificador,  
63                                         X = entradas,  
64                                         y = saídas,  
65                                         cv = 10,  
66                                         scoring = 'accuracy')
```

Em seguida executamos nosso classificador via KerasClassifier, passando nossa função valCruzada() como parâmetro, atualizando os registros a cada 10 rodadas, executando a rede 100 vezes. Logo após é feita de fato a validação cruzada, apenas recordando, esta etapa pega nossa

base de dados e divide em um número x de partes iguais e testa individualmente elas para que se verifique as devidas taxas de precisão e margem de erro.

```
68 mediaValCruzada = resultadoValCruzada.mean()  
69 desvioValCruzada = mediaValCruzada.std()
```

Por fim, executando as linhas de código acima temos as variáveis dedicadas aos resultados das execuções de nossa validação cruzada.

Nome	Tipo	Tamanho	Valor
desvioValCruzada	float64	1	0.0
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_
eteste	DataFrame	(143, 30)	Column names: radius_mean, texture_
etreino	DataFrame	(426, 30)	Column names: radius_mean, texture_
mediaValCruzada	float64	1	0.9156015037593985
resultadoValCruzada	float64	(10,)	[0.77192982 0.94736842 0.94736842 0.8... 0.94736842 ...]
saidas	DataFrame	(569, 1)	Column names: 0

No explorador de variáveis podemos conferir que como resultado de nossa validação cruzada agora temos uma taxa de precisão de 91%. Essa discrepância muito provavelmente está acontecendo por algum erro de leitura dos dados, já que agora, aplicando testes sobre partes individuais o percentual não só saiu do padrão anterior, mas aumentou consideravelmente.

Realizando Tuning do Modelo

Aqui para não continuar na mesmice do exemplo anterior, vamos fazer um processo bastante comum que é o reajuste manual dos parâmetros da rede, assim como os testes para se verificar qual configuração de parâmetros retorna melhores resultados. Até então o que eu lhe sugeri foi usar os parâmetros usuais assim como suas pré-configurações, apenas alterando as taxas de reajustes dos pesos e número de execuções da rede. Porém haverão situações onde este tipo de ajuste, na camada mais superficial da rede não surtirá impacto real, sendo necessário definir e testar parâmetros internos manualmente para se buscar melhores resultados. Consultando a documentação de nossas ferramentas podemos notar que existe uma série de parâmetros internos a nossa disposição. Como saber qual se adapta melhor a cada situação? Infelizmente em grande parte dos casos teremos de, de acordo com sua descrição na documentação, realizar testes com o mesmo inclusive de comparação com outros parâmetros da mesma categoria para encontrar o que no final das contas resultará em resultados de processamento mais íntegros.

Em algumas literaturas esse processo é chamado de tuning. Para também sair fora da rotina vamos aplicar tal processo a partir de um arquivo novo apenas com o essencial para o processo de tunagem. Outro ponto para por hora finalizar nossa introdução é que essa fase em especial costuma ser a mais demorada, uma vez que de acordo com o número de parâmetros a serem testados, cada teste é uma nova execução da rede neural.

Breast Cancer Dataset Tuning.py

```
1 import pandas as pd
2 import keras
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from keras.wrappers.scikit_learn import KerasClassifier
6 from sklearn.model_selection import GridSearchCV
```

Seguindo a mesma lógica de sempre, toda vez que criamos um novo arquivo de código, primeira etapa é realizar as devidas importações das bibliotecas e módulos necessários. Repare que na última linha existe uma ferramenta nova sendo importada, trata-se da GridSearchCV, do módulo model_selection da biblioteca sklearn. Em suma essa ferramenta permite de fato criarmos parâmetros manualmente para alimentar nossa rede neural, assim como passar mais de um parâmetro a ser testado para por fim serem escolhidos os que em conjunto mostraram melhores resultados de processamento.

```
8 entradas = pd.read_csv('entradas-breast.csv')
9 saídas = pd.read_csv('saídas-breast.csv')
```

Em seguida por meio da função .read_csv('') da biblioteca pandas importamos novamente nossas bases de dados de entrada e saída.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean, smoothness_mean, compactness_mean, concavity_mean, concave points_mean, symmetry_mean, fractal_dimension_mean, radius_se, texture_se, perimeter_se, area_se, smoothness_se, compactness_se, concavity_se, concave points_se, symmetry_se, fractal_dimension_se, radius_worst, texture_worst, perimeter_worst, area_worst, smoothness_worst, compactness_worst, concavity_worst, concave points_worst, symmetry_worst, fractal_dimension_worst
saídas	DataFrame	(569, 1)	Column names: 0

Se o processo ocorreu sem erros, as devidas variáveis foram criadas.

```
11 def tuningClassificador(optimizer, loss, kernel_initializer, activation, neurons):
12     classificadorTuning = Sequential()
13     classificadorTuning.add(Dense(units = neurons,
14                                    activation = activation,
15                                    kernel_initializer = kernel_initializer,
16                                    input_dim = 30))
17     classificadorTuning.add(Dropout(0.2))
18     classificadorTuning.add(Dense(units = neurons,
19                                    activation = activation,
20                                    kernel_initializer = kernel_initializer))
21     classificadorTuning.add(Dropout(0.2))
22     classificadorTuning.add(Dense(units = 1,
23                                    activation = 'sigmoid'))
24     classificadorTuning.compile(optimizer = optimizer,
25                                loss = loss,
26                                metrics = ['binary_accuracy'])
27
return classificadorTuning
```

Logo após já podemos criar nossa rede, dentro de uma função criada especificamente para ela de nome tuningClassificador, repare que até então não havíamos passado nenhum tipo de parâmetro em nossas funções, apenas seu bloco de código interno, dessa vez, como iremos definir os parâmetros manualmente, instanciados por uma variável, é interessante declará-los já como parâmetro da função. Em seguida criamos a mesma estrutura da rede neural anterior, sequencial, 30 neurônios na camada de entrada, 1 na camada de saída e dropouts entre elas, porém note que os parâmetros internos a cada camada agora possuem nomes genéricos para substituição. Por fim na camada de compilação também é mantido a métrica binary_accuracy uma vez que esta é uma classificação binária. Encerrando o bloco, apenas é dado o comando para retornar a própria função.

```
29 classificadorTunado = KerasClassifier(build_fn = tuningClassificador)
30
```

A seguir criamos uma variável de nome classificadorTunado, que recebe como atributo o KerasClassifier, que por sua vez tem em sua ativação nossa função tuningClassificador.

```
31 parametros = {'batch_size':[10,30],
32                 'epochs':[50,100],
33                 'optimizer':['adam','sgd'],
34                 'loss':['binary_crossentropy','hinge'],
35                 'kernel_initializer':['random_uniform','normal'],
36                 'activation':['relu','tanh'],
37                 'neurons':[10,8]}
```

Da mesma forma criamos uma nova variável, agora de nome parâmetros, que pela sintaxe recebe em forma de dicionário os parâmetros a serem instanciados em nossa rede assim como os valores a serem testados. Repare por exemplo que ‘activation’ quando instanciada, usará ‘relu’ em uma das suas fases de testes assim como ‘tanh’ em outra fase de teste, posteriormente a GridSearchCV irá verificar qual desses dois parâmetros retornou melhor resultado e irá nos sinalizar este.

```
39 tunagem = GridSearchCV(estimator = classificadorTunado,
40                         param_grid = parametros,
41                         scoring = 'accuracy')
```

Logo após criamos uma variável de nome tunagem, que recebe como atributo a ferramenta GridSearchCV, que por sua vez tem como parâmetros estimator = classificadorTunado, o que em outras palavras significa que ele usará o KerasClassifier rodando nossa função tuningClassificador sob os parâmetros, também tem como parâmetro param_grid = parametros, o que significa que o classificador irá receber os parâmetros que definimos manualmente na variável parametros, por fim scoring = ‘accuracy’ que simplesmente estará aplicando métodos focados em precisão.

```
43 tunagem = tunagem.fit(entradas,saidas)
44
```

Por fim, criamos uma variável de nome tunagem, que aplica sobre si a função .fit() passando para a mesma como parâmetros entradas e saídas. Como das outras vezes, ao selecionar essa linha de código e executar, a função .fit() dará início a execução da rede neural. Como dito anteriormente, este processo de testar todos os parâmetros buscando os melhores é bastante demorado, dependendo muito do hardware de sua máquina, dependendo de sua configuração, esse processo pode literalmente demorar algumas horas.

```
binary_accuracy: 0.9686
Epoch 96/100
569/569 [=====] - 1s 2ms/step - loss: 0.2431 -
binary_accuracy: 0.9663
Epoch 97/100
569/569 [=====] - 1s 2ms/step - loss: 0.2124 -
binary_accuracy: 0.9504
Epoch 98/100
569/569 [=====] - 1s 2ms/step - loss: 0.2342 -
binary_accuracy: 0.9663
Epoch 99/100
569/569 [=====] - 1s 2ms/step - loss: 0.2219 -
binary_accuracy: 0.9651
Epoch 100/100
569/569 [=====] - 1s 2ms/step - loss: 0.2393 -
binary_accuracy: 0.9716
```

Acompanhando o processo via console você de fato irá notar que essa rede será processada e reprocessada muitas vezes, cada vez voltando à estaca zero e repetindo todo o processo com os parâmetros definidos anteriormente.

```
45 melhores_parametros = tunagem.best_params_
46 melhor_margem_precisao = tunagem.best_score_
```

Para finalizar o processo de tuning, podemos como de costume usar funções que nos mostrem de forma mais clara os resultados. Inicialmente criamos uma variável de nome melhores_parametros que por fim aplica sobre tunagem o método best_params_, por fim criamos uma variável de nome melhor_margem_precisao que sobre tunagem aplica o método best_score_. Selecionando e executando esse bloco de código temos acesso a tais dados.

Chave	Tipo	Tamanho	Valor
activation	str	1	relu
batch_size	int	1	10
epochs	int	1	100
kernel_initializer	str	1	random_uniform
loss	str	1	binary_crossentropy
neurons	int	1	8
optimizer	str	1	adam

Salvar e Fechar **Fechar**

Via explorador de variáveis você pode verificar quais os parâmetros que foram selecionados quanto a sua margem de precisão. Uma vez que você descobriu quais são os melhores parâmetros você pode voltar ao seu código original, fazer as devidas substituições e rodar novamente sua rede neural artificial.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture ..
melhor_margem_precisao	float64	1	0.9780667838312829
melhores_parametros	dict	7	{'activation':'relu', 'batch_size':10, 'epoch ..}
parametros	dict	7	{'batch_size':[10, 30], 'epochs':[100], 'optimizer':['adam', 'sgd'] ..}
saidas	DataFrame	(569, 1)	Column names: 0

Por fim também está disponível para visualização a variável dedicada a retornar o percentual de acerto de nosso processo de tuning. Assim como o valor aproximado exibido no console no

processo de testes, agora podemos ver de forma clara que usando dos melhores parâmetros essa rede chega a uma margem de precisão de 97% em sua classificação binária.

Código Completo:

```
1 import pandas as pd
2 import keras
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from keras.wrappers.scikit_learn import KerasClassifier
6 from sklearn.model_selection import GridSearchCV
7
8 entradas = pd.read_csv('entradas-breast.csv')
9 saidas = pd.read_csv('saidas-breast.csv')
10
11 def tuningClassificador(optimizer, loss, kernel_initializer, activation, neurons):
12     classificadorTuning = Sequential()
13     classificadorTuning.add(Dense(units = neurons,
14                                 activation = activation,
15                                 kernel_initializer = kernel_initializer,
16                                 input_dim = 30))
17     classificadorTuning.add(Dropout(0.2))
18     classificadorTuning.add(Dense(units = neurons,
19                                 activation = activation,
20                                 kernel_initializer = kernel_initializer))
21     classificadorTuning.add(Dropout(0.2))
22     classificadorTuning.add(Dense(units = 1,
23                                 activation = 'sigmoid'))
24     classificadorTuning.compile(optimizer = optimizer,
25                                 loss = loss,
26                                 metrics = ['binary_accuracy'])
27     return classificadorTuning
28
29 classificadorTunado = KerasClassifier(build_fn = tuningClassificador)
30 parametros = {'batch_size': [10,30],
31               'epochs': [50,100],
32               'optimizer': ['adam', 'sgd'],
33               'loss': ['binary_crossentropy', 'hinge'],
34               'kernel_initializer': ['random_uniform', 'normal'],
35               'activation': ['relu', 'tanh'],
36               'neurons': [10,8]}
37 tunagem = GridSearchCV(estimator = classificadorTunado,
38                        param_grid = parametros,
39                        scoring = 'accuracy')
40 tunagem = tunagem.fit(entradas,saidas)
41 melhores_parametros = tunagem.best_params_
42 melhor_margem_precisao = tunagem.best_score_
```

Realizando Testes Sobre Uma Amostra

Uma das propriedades importantes de uma rede neural após construída, treinada e testada é que a partir deste ponto podemos salvar esse modelo para reutilização. Uma das práticas mais comuns em uma rede como esta, de classificação binária, é a partir do modelo pronto e treinado reutilizar a mesma para que se teste alguma amostra individualmente. Seguindo a linha de raciocínio deste exemplo, lembre-se que aqui estamos passando uma série de atributos para que seja verificado se naquela imagem de mamografia o resultado é positivo para câncer ou negativo para câncer. Dessa forma, podemos pegar uma amostra individual de nossa base de dados e aplicar o teste da rede neural sobre a mesma.

```

Breast Cancer Dataset Teste Uma Amostra.py

1 import pandas as pd
2 import numpy as np
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout

```

A partir de um novo arquivo iniciamos como sempre com as importações necessárias.

```

6 entradas = pd.read_csv('entradas-breast.csv')
7 saídas = pd.read_csv('saídas-breast.csv')

```

Realizamos a importação de nossa base de dados, dividida em dados de entrada e de saída a partir dos seus respectivos arquivos do tipo .csv.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ...
saídas	DataFrame	(569, 1)	Column names: 0

Como de praxe realizamos a verificação se não houve nenhum erro no processo tanto no processo de importação quanto de criação das respectivas variáveis.

```

9 classificadorA = Sequential()
10 classificadorA.add(Dense(units = 8,
11                         activation = 'relu',
12                         kernel_initializer = 'normal',
13                         input_dim = 30))
14 classificadorA.add(Dropout(0.2))
15 classificadorA.add(Dense(units = 8,
16                         activation = 'relu',
17                         kernel_initializer = 'normal'))
18 classificadorA.add(Dropout(0.2))
19 classificadorA.add(Dense(units = 1,
20                         activation = 'sigmoid'))
21 classificadorA.compile(optimizer = 'adam',
22                         loss = 'binary_crossentropy',
23                         metrics = ['binary_accuracy'])
24 classificadorA.fit(entradas,
25                     saídas,
26                     batch_size = 10,
27                     epochs = 100)

```

Criamos como usualmente fizemos anteriormente a estrutura de nossa rede neural, agora já parametrizada com os melhores parâmetros encontrados no processo de tuning.

```

Epoch 96/100
569/569 [=====] - 0s 128us/step - loss: 0.3394 -
binary_accuracy: 0.8541
Epoch 97/100
569/569 [=====] - 0s 126us/step - loss: 0.3236 -
binary_accuracy: 0.8594
Epoch 98/100
569/569 [=====] - 0s 130us/step - loss: 0.3346 -
binary_accuracy: 0.8401
Epoch 99/100
569/569 [=====] - 0s 135us/step - loss: 0.3218 -
binary_accuracy: 0.8699
Epoch 100/100
569/569 [=====] - 0s 125us/step - loss: 0.3339 -
binary_accuracy: 0.8506
Out[2]: <keras.callbacks.History at 0x1bedba79cf8>

```

Selecionando e executando todo bloco de código de nossa rede neural acompanhamos via console o processo de execução da mesma assim como se houve algum erro no processo. Lembrando que a esta altura o mais comum é se houver erros, estes ser erros de sintaxe, pois da estrutura lógica da rede até aqui você já deve compreender.

```
29 objeto = np.array([[17.99,10.38,122.8,1001,0.1184,0.2776,0.3001,
30           0.1471,0.2419,0.07871,1095,0.9053,8589,153.4,
31           0.006399,0.04904,0.05373,0.01587,0.03003,0.006193,
32           25.38,17.33,184.6,2019,0.1622,0.6656,0.7119,0.2654,
33           0.4601,0.1189]])
```

Dando sequência, vamos ao que realmente interessa. Uma vez feito todo processo de importação das bibliotecas, módulos, base de dados e executada a rede neural uma vez para seu respectivo treino, hora de aplicar teste sobre uma amostra. Para isso, simplesmente criamos uma variável de nome objeto que recebe como atributo uma array do tipo numpy onde selecionamos, aqui nesse exemplo, uma linha aleatória em nossa base de dados com seus respectivos 30 atributos previsores.

```
35 previsorA = classificadorA.predict(objeto)
36 previsorB = (previsorA > 0.5)
```

Por fim criamos duas variáveis, uma de nome previsorA que recebe como atributo a função .predict() parametrizada com nosso objeto criado anteriormente, executando tudo sobre nossa rede neural. Da mesma forma, criamos uma variável de nome previsorB que simplesmente irá conferir se previsorA for maior do que 0.5, automaticamente essa expressão nos retornará um dado True ou False.

Nome	Tipo	Tamanho	Valor
entradas	DataFrame	(569, 30)	Column names: radius_mean, texture_mean, perimeter_mean, area_mean ... [[1.799e+01 1.038e+01 1.228e+02 ... 2.654e-01 4.601e-01 1.189e-01]]
objeto	float64	(1, 30)	[[3.970654e-06]]
previsorA	float32	(1, 1)	[[False]]
previsorB	bool	(1, 1)	[[False]]
saídas	DataFrame	(569, 1)	Column names: 0

De fato, executando todo último bloco de código são criadas as respectivas variáveis. A variável previsorA nos deu como retorno um número extremamente pequeno, porém, como mencionado anteriormente, o ideal é expormos nossas informações de forma clara. O previsorB por sua vez, com base em previsorA nos retorna False, que em outras palavras significa que para aquela amostra o resultado é negativo para câncer.

Salvando o Modelo Para Reutilização

Como dito anteriormente, uma prática comum é a reutilização de um modelo de rede uma vez que esse está pronto, sem erros de código, treinado já com os melhores parâmetros, para que ele possa ser ferramenta para solucionar certos problemas computacionais de mesmo tipo. Isto pode ser feito de forma bastante simples, salvando o código da rede neural (obviamente) assim como sua estrutura funcional e pesos por meio de arquivos do tipo .py, .json e .h5. Raciocine que em bases de dados pequenas, como esta que estamos usando para testes, o processamento dela pela rede se dá em poucos segundos, porém, em determinadas situações reais é interessante você

ter uma rede pronta sem a necessidade de ser treinada novamente (ainda mais com casos onde são usadas bases com milhões de dados como em big data).

```
1 import pandas as pd
2 import numpy as np
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5
6 entradas = pd.read_csv('entradas-breast.csv')
7 saidas = pd.read_csv('saidas-breast.csv')
8
9 classificadorB = Sequential()
10 classificadorB.add(Dense(units = 8,
11                         activation = 'relu',
12                         kernel_initializer = 'normal',
13                         input_dim = 30))
14 classificadorB.add(Dropout(0.2))
15 classificadorB.add(Dense(units = 8,
16                         activation = 'relu',
17                         kernel_initializer = 'normal'))
18 classificadorB.add(Dropout(0.2))
19 classificadorB.add(Dense(units = 1,
20                         activation = 'sigmoid'))
21 classificadorB.compile(optimizer = 'adam',
22                         loss = 'binary_crossentropy',
23                         metrics = ['binary_accuracy'])
24 classificadorB.fit(entradas,
25                     saidas,
26                     batch_size = 10,
27                     epochs = 100)
```

Repare que esta é simplesmente a mesma estrutura criada anteriormente, já com as devidas importações, já com toda estrutura da rede neural, etc... Simplesmente você pode reaproveitar todo esse bloco de código em um novo arquivo para poupar seu tempo.

```
29 classificador_json = classificadorB.to_json()
30
31 with open('classificador_binario.json', 'w') as json_file:
32     json_file.write(classificador_json)
33
34 classificadorB.save_weights('classificador_binario_pesos.h5')
```

O processo em si, de salvar nosso modelo para reutilização basicamente de dá da seguinte forma. Inicialmente criamos uma variável de nome classificador_json que recebe toda estrutura de nossa rede neural classificadorB com a função `.to_json()`, essa função permite salvar esse modelo neste formato de arquivo nativo para as IDEs Python. Em seguida entramos com o método `with open()` que por sua vez recebe como parâmetro inicialmente ‘`classificador_binario.json`’, ‘`w`’, isto nada mais é do que o comando que estamos dando para que se crie um arquivo com o nome `classificador_binario.json` na mesma pasta onde estão seus arquivos de código, na sequência existe a condição `as json_file:` que permite definirmos algum atributo interno. Dentro de `json_file` simplesmente aplicamos sobre si a função `.write()` passando como parâmetro todo o conteúdo de `classificador_json`. Por fim, fazemos um processo parecido para salvar os pesos, mas agora, usando uma função interna da biblioteca pandas mesmo, então, aplicando sobre `classificadorB` a função `.save_weights()` e definindo como parâmetro também o nome do arquivo que guardará esses pesos, no formato `h5`. Selecionando e rodando esse bloco de código serão criados os respectivos arquivos na pasta onde está o seu código.

Nome	Tipo	Tamanho
saídas-breast.csv	Microsoft Excel ...	2 KB
Breast Cancer Dataset.py	Arquivo PY	5 KB
classificador_binario.json	Arquivo JSON	2 KB
classificador_binario_pesos.h5	Arquivo H5	18 KB
entradas-breast.csv	Microsoft Excel ...	115 KB

Carregando Uma Rede Neural Artificial Pronta Para Uso

```
1 import numpy as np
2 from keras.models import model_from_json
```

Como sempre, começando pelas importações, note que há a importação de uma ferramenta ainda não mencionada nos exemplos anteriores. A ferramenta `model_from_json`, como o próprio nome indica, serve para importar modelos prontos, desde que estejam no formato json, esta ferramenta faz parte do módulo `models` da biblioteca `keras`.

```
4 arquivo = open('classificador_binario.json', 'r')
5 estrutura_rede = arquivo.read()
6 arquivo.close()
```

Em seguida criamos uma variável de nome `arquivo`, que recebe como atributo o método `open`, parametrizado com o nome de nosso arquivo salvo anteriormente em forma de string, assim como o segundo parâmetro '`r`', uma vez que agora estamos fazendo a leitura do arquivo. Na sequência criamos uma variável de nome `estrutura_rede` que recebe os dados de arquivo pela função `.read()`. Por fim, podemos fechar o arquivo carregado por meio da função `.close()`. Pode ficar tranquilo que uma vez feita a leitura ela ficará pré-alocada em memória para nosso interpretador, o arquivo em si pode ser fechado normalmente.

Nome	Tipo	Tamanho	Valor
estrutura_rede	str	1	{"class_name": "Sequential", "config": {"name": "sequential_2", "layer ...

Se o processo de leitura a partir do arquivo json foi feita corretamente, a respectiva variável será criada.



The screenshot shows a window titled "Editor de texto - estrutura_rede". The content of the editor is a JSON object representing a neural network structure. The JSON is as follows:

```
{"class_name": "Sequential", "config": {"name": "sequential_2", "layers": [{"class_name": "Dense", "config": {"name": "dense_4", "trainable": true, "batch_input_shape": [null, 30], "dtype": "float32", "units": 8, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name": "RandomNormal", "config": {"mean": 0.0, "stddev": 0.05, "seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}, {"class_name": "Dropout", "config": {"name": "dropout_3", "trainable": true, "rate": 0.2, "noise_shape": null, "seed": null}, {"class_name": "Dense", "config": {"name": "dense_5", "trainable": true, "units": 8, "activation": "relu", "use_bias": true, "kernel_initializer": {"class_name": "RandomNormal", "config": {"mean": 0.0, "stddev": 0.05, "seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}, {"class_name": "Dropout", "config": {"name": "dropout_4", "trainable": true, "rate": 0.2, "noise_shape": null, "seed": null}, {"class_name": "Dense", "config": {"name": "dense_6", "trainable": true, "units": 1, "activation": "sigmoid", "use_bias": true, "kernel_initializer": {"class_name": "VarianceScaling", "config": {"scale": 1.0, "mode": "fan_avg", "distribution": "uniform", "seed": null}}, "bias_initializer": {"class_name": "Zeros", "config": {}}, "kernel_regularizer": null, "bias_regularizer": null, "activity_regularizer": null, "kernel_constraint": null, "bias_constraint": null}}]}, "keras_version": "2.2.4", "backend": "tensorflow"}]
```

At the bottom of the editor window, there are two buttons: "Salvar e Fechar" (Save and Close) and "Fechar" (Close).

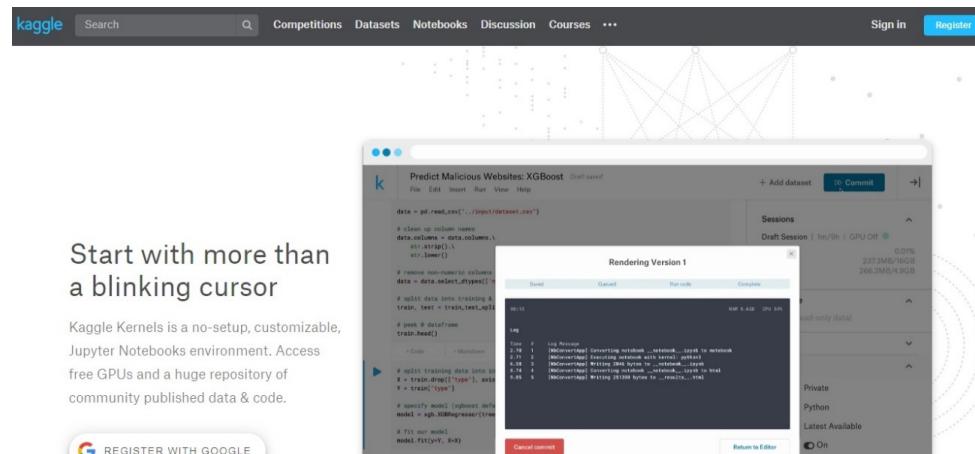
Abrindo a variável `estrutura_rede` é possível ver que de fato todos os parâmetros de nossa rede neural estão salvos de forma que nosso interpretador conseguirá fazer a leitura e uso dos mesmos. A partir daqui, é possível criar qualquer modelo previsor, instanciando nosso classificador.

Regressão de Dados de Planilhas Excel – Autos Dataset

Uma vez entendidos os conceitos básicos do que é uma rede neural artificial e sua estrutura lógica e de código quando o problema computacional proposto é uma classificação de dados, hora de gradualmente avançarmos para outros modelos, não necessariamente de maior complexidade, mas modelos que elucidarão as propostas de outros tipos de redes neurais desenvolvidas para outros fins.

Um dos tipos de redes neurais artificiais mais comumente utilizados são as redes que aplicam métodos de regressão. Em outras palavras, com base em uma grande base de dados, treinaremos uma rede para reconhecer padrões que permitirão encontrar por exemplo, o valor médio de um automóvel usado se comparado a outros de mesmo tipo.

Aqui como exemplo usaremos o Autos Dataset, uma base de dados com milhares de carros usados catalogados como em uma revenda. Aproveitaremos esse modelo para ver como se dá o processo de polimento dos dados para criarmos uma base com valores mais íntegros, capazes de gerar resultados mais precisos.



Start with more than
a blinking cursor

Kaggle Kernels is a no-setup, customizable,
Jupyter Notebooks environment. Access
free GPUs and a huge repository of
community published data & code.

REGISTER WITH GOOGLE

A base de dados que iremos usar neste exemplo pode ser baixada gratuitamente no Kaggle, um repositório de machine learning parecido com o ACI, porém mais diversificado e com suporte de toda uma comunidade de desenvolvedores.

Dataset

Used cars database

Over 370,000 used cars scraped from Ebay Kleinanzeigen

Orges Leka • updated 3 years ago (Version 3)

Data Kernels (224) Discussion (9) Activity Metadata Download (18 MB) New Notebook

Usability 8.5 License CCO: Public Domain Tags automobiles

Description

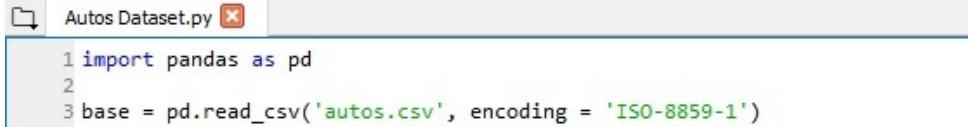
Over 370000 used cars scraped with Scrapy from Ebay-Kleinanzeigen. The content of the data is in german, so one has to translate it first if one can not speak german. Those fields are included: autos.csv:

- dateCrawled : when this ad was first crawled, all field-values are taken from this date
- name : "name" of the car

Usando a ferramenta de busca do site você pode procurar por Used cars database. O dataset em si, como exibido na imagem acima, trata-se de uma base de dados de aproximadamente 370 mil registros de anúncios de carros usados retirado dos classificados do Ebay. Todo o que precisamos fazer é o download da base de dados no formato .csv.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	dateCrawled	ed.name	seller	offerType	price	abtest	vehicleType	year	Registration	gearbox	powerPs	model	kilometer	monthOfRegistration	fuelType	brand	notRepairedDamage	dateCreated	nrOfPictures
2	2016-03-24 11:52:17	Golf_3_1.6	privat	Angebot	480	test	1993	manuell	0,goif	150000	0,benzin	volkswagen	,kilometer	2016-03-24 00:00:00,0	70435	2016-04-07 03:16:57			
3	2016-03-24 10:58:45	AS_Sportback	2.7	Tdi	privat	Angebot	18300	test,suv	2011	manuell	190	,125000	5,diesel	audi	ja	2016-03-24 00:00:00,0	66954	2016-04-07 01:46:50	
4	2016-03-14 12:52:21	Jeep_Grand_Cherokee	"Overland"	privat	Angebot	9800	test,suv	2004	automatik	163	grand	125000	0,diesel	jeep	ja	2016-03-14 00:00:00,0	90408	2016-04-05 12:47:46	
5	2016-03-17 16:54:04	GOLF_4_1.4_TDI	PD_Classic	privat	Angebot	1500	test,leinwagen	2001	manuell	75	goif	150000	6,benzin	volkswagen	nein	2016-03-17 00:00:00,0	91074	2016-03-17 17:40:17	
6	2016-03-31 17:25:20	Skoda_Fabia_1.4_TDI	PD_Classic	privat	Angebot	3600	test,leinwagen	2008	manuell	69	fabia	90000	7,diesel	skoda	nein	2016-03-31 00:00:00,0	60437	2016-04-06 10:17:21	
7	2016-04-04 17:36:23	BMW_316i_e36_Limousine	"Bastlerfahrzeug_Export"	privat	Angebot	650	test,limousine	1995	manuell	102	3er	150000	10,benzin	bmw	ja	2016-04-04 00:00:00,0	33775	2016-04-06 19:17:0	
8	2016-04-01 20:48:51	Peugeot_208_CC_110_Platinum	privat	Angebot	220	test,cabrio	2004	manuell	109	2	reihen	150000	8,benzin	peugeot	nein	2016-04-01 00:00:00,0	67112	2016-04-05 18:18:39	
9	2016-03-21 18:54:38	VW_Derby_BJ_80_Scheunenfund	privat	Angebot	0	test,limousine	1980	manuell	50	andere	40000	7,benzin	vw	volkswagen	nein	2016-03-21 00:00:00,0	13348	2016-03-25 16:47:58	
10	2016-04-04 23:42:13	Ford_C-Max_Titanium_1.0_EcoBoost	privat	Angebot	14500	contr,bus	2014	manuell	125	c_max	30000	8,benzin	ford	2016-04-04 00:00:00,0	94950	2016-04-04 23:42:13			
11	2016-03-17 10:53:30	VW_Golf_4_1.4_tuerig_zu_verkaufen_mit_Anhängerkupplung	privat	Angebot	599	test,leinwagen	1998	manuell	101	goif	150000	0,volkswagen	,2016-03-17 00:00:00,0	2,7472	2016-03-31 17:17				
12	2016-03-26 19:54:18	Mazda_3_1.6_Sport	privat	Angebot	2000	control,limousine	2004	manuell	105	3	reihen	150000	12,benzin	mazda	nein	2016-03-26 00:00:00,0	96224	2016-04-06 10:45:34	
13	2016-04-07 10:06:22	Volkswagen_Passat_2.0_TDI_Confortline	privat	Angebot	2799	control,kombi	2005	manuell	140	passat	150000	12,diesel	volkswagen	ja	2016-04-07 00:00:00,0	57290	2016-04-07 10:22		
14	2016-03-15 22:49:09	VW_Passat_Facelift_35_75Tzter	privat	Angebot	999	control,kombi	1995	manuell	115	passat	150000	11,benzin	volkswagen	,2016-03-15 00:00:00,0	3,7269	2016-04-01 13:16:16			
15	2016-03-21 21:37:40	VW_PASSAT_1.9_TDI_131PS_LEDER	privat	Angebot	2500	control,kombi	2004	manuell	131	passat	150000	2,volkswagen	nein	2016-03-21 00:00:00,0	90762	2016-03-23 02:50:25			
16	2016-03-21 12:57:01	Nissan_Navara_2.5DPR_4WD_SEA4_XKlima_Sitzheizg_Bluetooth_DoppelKabine	privat	Angebot	17999	control,suv	2011	manuell	190	navara	70000	3,diesel	nissan	nein	2016-03-21 00:00:00,0	0,04177			
17	2016-03-11 21:39:15	KA_Lufthansa_Edition_450K_V8	privat	Angebot	450	test,leinwagen	1910	0,ka	5000	0,benzin	ford	2016-03-11 00:00:00,0	24148	2016-03-19 08:46:47					
18	2016-04-01 12:46:46	Polo_6n_1.4_privat	privat	Angebot	300	test	2016	6n	10	polo	150000	0,benzin	volkswagen	,2016-04-01 00:00:00,0	3,8871	2016-04-01 12:46:46			
19	2016-03-20 10:25:19	Renault_Twingo_1.2_16V_Aut.	privat	Angebot	1750	control,leinwagen	2014	aut,motif	75	twingo	150000	2,benzin	renault	nein	2016-03-20 00:00:00,0	65559	2016-04-06 13:16:07		
20	2016-03-23 15:48:05	Ford_C-MAX_2.0_TDCI_DPF_Titanium	privat	Angebot	750	test,bus	2007	manuell	136	c_max	150000	6,diesel	ford	nein	2016-03-23 00:00:00,0	88361	2016-04-05 18:45:11		
21	2016-04-01 22:55:47	Mercedes_Benz_A_160_Classic_Klima	privat	Angebot	1850	test,bus	2004	manuell	102	a_klasse	150000	0,benzin,mercedes_benz	nein	2016-04-01 00:00:00,0	49565	2016-04-05 22:46:05			
22	2016-04-01 19:56:48	Volkswagen_Scirocco_1.4_TSI_Sport	privat	Angebot	10400	control,coupe	2009	manuell	160	scirocco	100000	4,benzin	volkswagen	nein	2016-04-01 00:00:00,0	75365	2016-04-05 16:45:49		
23	2016-03-27 11:38:00	BMW_530i_TUV_7/17_Scheckheftgepflegt_sehr_guter_Zustand	privat	Angebot	3699	test,limousine	2002	automatik	231	5er	150000	7,benzin	bmw	nein	2016-03-27 00:00:00,0	68309	2016-04-04 10:22:57		

Abrindo o arquivo pelo próprio Excel pode-se ver que é uma daquelas típicas bases de dados encontradas na internet, no sentido de haver muita informação faltando, muitos dados inconsistentes, muito a ser pré-processado antes de realmente começar a trabalhar em cima dessa base.



```

import pandas as pd
base = pd.read_csv('autos.csv', encoding = 'ISO-8859-1')

```

Dando início ao que interessa, como sempre, todo código começa com as devidas importações das bibliotecas e módulos necessários, por hora, simplesmente importamos a biblioteca pandas para que possamos trabalhar com arrays. Em seguida já criamos uma variável de nome base que pela função `.read_csv()` realiza a leitura e importação dos dados de nosso arquivo `autos.csv`, note que aqui há um parâmetro adicional `encoding = 'ISO-8859-1'`, este parâmetro nesse caso se faz necessário em função de ser uma base de dados codificada em alguns padrões europeus, para não haver nenhum erro de leitura é necessário especificar a codificação do arquivo para o interpretador.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerType, price, abtest, veh ...

Feita a importação da base de dados, podemos conferir em nosso explorador de variáveis que o objeto foi importado e reconhecido como um DataFrame, contendo 371528 amostras com 20 atributos, respectivamente.

base - DataFrame

Index	dateCrawled	name	seller	offerType	price	abtest	vehicleType	/yearO ^
0	2016-03-24 11:52:17	Golf_3_1.6	privat	Angebot	480	test	nan	1993
1	2016-03-24 10:58:45	A5_Sportback...	privat	Angebot	18300	test	coupe	2011
2	2016-03-14 12:52:21	Jeep_Grand_C...	privat	Angebot	9800	test	suv	2004
3	2016-03-17 16:54:04	GOLF_4_1_4_...	privat	Angebot	1500	test	kleinwag...	2001
4	2016-03-31 17:25:20	Skoda_Fabia_...	privat	Angebot	3600	test	kleinwag...	2008
5	2016-04-04 17:36:23	BMW_316i_e...	privat	Angebot	650	test	limousine	1995
6	2016-04-01 20:48:51	Peugeot_206_...	privat	Angebot	2200	test	cabrio	2004
7	2016-03-21 18:54:38	VW_Derby_Bj...	privat	Angebot	0	test	limousine	1980
8	2016-04-04 23:42:13	Ford_C_Max...	privat	Angebot	14500	con...	bus	2014
9	2016-03-17 10:53:50	VW_Golf_4_5_...	privat	Angebot	999	test	kleinwag...	1998
10	2016-03-26 19:54:18	Mazda_3_1.6_...	privat	Angebot	2000	con...	limousine	2004
11	2016-04-07 10:06:22	Volkswagen_P...	privat	Angebot	2799	con...	kombi	2005
12	2016-03-15 22:49:09	VW_Passat_Fa...	privat	Angebot	999	con...	kombi	1995
13	2016-03-21	VW PASSAT 1...	privat	Angebot	2500	con...	kombi	2004

Formato Redimensionar Cor de fundo Min/max de coluna Salvar e Fechar Fechar

Abrindo a variável para uma rápida análise visual é possível ver que existem muitos dados faltando, ou inconsistentes, que literalmente só atrapalharão nosso processamento. Sendo assim, nessa fase inicial de polimento desses dados vamos remover todos os atributos (todas as colunas) de dados que não interessam à rede.

```
5 base = base.drop('dateCrawled', axis=1)
6
```

Aplicando diretamente sobre nossa variável base a função `.drop()` passando como parâmetro o nome da coluna e seu eixo, podemos eliminá-la sem maiores complicações. Lembrando apenas que o parâmetro `axis` quando definido com valor 0 irá aplicar a função sobre uma linha, enquanto valor 1 aplicará a função sobre uma coluna.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerTyp... price, abtest, veh ...

Base original.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 20)	Column names: dateCrawled, name, seller, offerTyp... price, abtest, veh ...

Após execução da linha de código anterior. Note que eliminamos a coluna ‘dateCrawled’, e como aplicamos a função sobre nossa variável, eliminamos essa coluna de forma permanente.

```

5 base = base.drop('dateCrawled', axis=1)
6 base = base.drop('dateCreated', axis=1)
7 base = base.drop('nrOfPictures', axis=1)
8 base = base.drop('postalCode', axis=1)
9 base = base.drop('lastSeen', axis=1)

```

Sendo assim, por meio da função `.drop()` podemos remover de imediato as colunas que não fazem sentido manter em nossa base de dados, em seguida, podemos verificar se uma ou mais colunas pode ser eliminada em relação a inconsistência de seus dados.

```

11 base['name'].value_counts()
12

```

Para tal feito selecionamos manualmente o nome de uma coluna seguido da função `.value_counts()` que irá apresentar esses dados agrupados quanto a igualdade.

In [37]: base['name'].value_counts()	Out[37]:
Ford_Fiesta	657
BMW_318i	627
Opel_Corsa	622
Volkswagen_Golf_1.4	603
BMW_316i	523
BMW_320i	492
Volkswagen_Polo	475
Renault_Twingo	447
Volkswagen_Golf	428
Volkswagen_Golf_1.6	413
Volkswagen_Polo_1.2	412
BMW_116i	394
Opel_Corsa_1.2_16V	373
Opel_Corsa_B	369
Opel_Astra	366

Acompanhando via console, podemos notar que, por exemplo Ford_Fiesta aparece 657 vezes em nossa base de dados, o problema é que o interpretador está fazendo esse levantamento de forma literal, existem 657 “Ford_Fiesta” assim como devem existir outras dezenas ou centenas de “Ford Fiesta”. Esse tipo de inconsistência é muito comum ocorrer quando importamos de plataformas onde é o usuário que alimenta com os dados, em função de cada usuário cadastrar seu objeto/item sem uma padronização de sintaxe.

In [38]:	Out[38]:
Ford_Mondeo_2.0_Kombi_Ghia_*Xenon_Klima*	1
Grand_Cherokee_2.7_CRD_GÜRNE_4_PM1!_2017/05TÜV	1
Volvo_V40_Tuev_neu	1
Mazda_121_Mit_TÜV	1
Seat_Altea_1.4_TSI_XL_Reference_Comfort	1
Schicker_Gelaendewagen	1
Verkaufe_Audi_a4_b5_Facelift	1
Audi_TT_Tuev_Neu_Service_Neu!!!!	1
Auto_zu_verkaufen	1
Ford_Focus_2.0_TDCi_DPF_Style_Navi	1
Opel_Rekord_Olympia_P2_Coupe_Oldtimer/_Echter_Scheunenfund	1
Chrysler_Status_2_0_lx_Bastler_Schlachten_Leder	1
BMW_320i_MP3_PDC_Regensor_SCHECKHEFT	1
Mercedes_Benz_A_140_Facelift_Md_04_nur_100tkm_TÜV_2.Hd.!	1
Mercedes_C_Coupe_51tkm_aus_1ter_Hand	1
Name: name, Length: 233531, dtype: int64	

Da mesma forma repare, por exemplo, que existe apenas uma amostra de Audi_TT_Tuev_Neu_Service_Neu!!!!, isso se dá pela forma da escrita do nome do objeto, de

fato, ninguém mais iria anunciar seu Audi TT escrevendo aquele nome exatamente daquela forma. Sendo assim, a coluna ‘names’ pode ser eliminada em função de sua inconsistência.

```
13 base = base.drop('name', axis=1)
14 base = base.drop('seller', axis=1)
15 base = base.drop('offerType', axis=1)
```

Nesta etapa, eliminamos três colunas em função de suas inconsistências em seus dados.

```
17 varTeste1 = base.loc[base.price <= 10]
18
```

Outro processo comumente realizado é tentarmos filtrar e eliminar de nossa base de dados amostras/registros com valores absurdos. Aqui como exemplo inicial criamos uma variável de nome varTeste1 que recebe como atributo base com o método .loc onde queremos descobrir de nossa base, no atributo price, todas amostras que tem valor igual ou menor que 10. Lembre-se que estamos trabalhando em uma base de dados de carros usados, é simplesmente impossível haver carros com preço igual ou menor a 10 Euros, com certeza isso é inconsistência de dados também.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(371528, 15)	Column names: name, seller, offerType, price abtest, vehicleType, yea ...
varTeste1	DataFrame	(12118, 15)	Column names: name, seller, offerType, price abtest, vehicleType, yea ...

Selecionando a linha de código anterior e a executando note que é criada a variável varTeste1, repare que existem 12118 amostras com valor igual ou menor que 10, tantas amostras com valores tão errados certamente iriam atrapalhar o processamento de nossa rede neural posteriormente.

```
19 base = base[base.price > 10]
20
```

Aplicando este método base.price > 10 diretamente em nossa base, fazemos com que a mesma seja atualizada mantendo apenas os valores de registros com o atributo ‘price’ maior que 10.

```
19 varTeste2 = base.loc[base.price > 350000]
20 base = base[base.price < 350000]
```

Da mesma forma, filtramos e eliminamos de nossa base de dados todos registros com respectivo preço maior que 350000. Totalizando 115 amostras eliminadas da base.

```
22 base.loc[pd.isnull(base['vehicleType'])]
23 base['vehicleType'].value_counts()
```

Para finalizar nossa fase de polimento dos dados, outro processo bastante comum é corrigir os valores que estiverem literalmente faltando em nossa base de dados. Por meio do método .loc e da função .isnull() sobre nossa base, passando como parâmetro ‘vehicleType’, estaremos fazendo este tipo de verificação. Assim como visto anteriormente que pela função .value_counts() iremos ver esses dados agrupados quanto a sua nomenclatura.

```
In [ . ]: base.loc[pd.isnull(base['vehicleType'])]
Out[ . ]:
```

		name	.notRepairedDamage
0		Golf_3_1.6	.
16		Polo_6n_1_4	.
22		Opel_Meriva_1.Hand_TÜV_3.2018	.
26		Citroen_C4_Grand_Picasso.	.
31		Renault_clio_1.2_TÜV_07/2016	.
35		VW_Golf_3	.
37		Renault_Kangoo_1.9_Diesel	.
48		VW_Golf_6__Klima__Alu__Scheckheft_!!!	.
51		Fiat_punto_5_tuerer_6_gang	.
52		Verkaufe_meinen_kleinen_wegen_neu_Anshaffung	.
58		Seat_inca_1.9SDI__LKW_Zulassung__TÜV_NEU	.
66		Opel_Astra_1.4_mit_vielnen_Extras!!!!	.
72		BMW_520i_E39	.
78		Golf_4_TDi_1.9._3_tuerig	.
81		Opel_Astra_F_Cabrio	.

Via console podemos acompanhar o retorno da função `.isnull()`. Muito cuidado para não confundir a nomenclatura, neste caso, ja e nein significam sim e não, respectivamente, enquanto NaN é uma abreviação gerada pelo nosso interpretador quando realmente nesta posição da lista/dicionário existem dados faltando. Em outras palavras, ja e nein são dados normais, presentes na classificação, NaN nos mostra que ali não há dado nenhum.

```
In [ . ]: base['vehicleType'].value_counts()
Out[ . ]:
```

vehicleType	count
limousine	93614
kleinwagen	78014
kombi	65921
bus	29699
cabrio	22509
coupe	18386
suv	14477
andere	3125

Name: vehicleType, dtype: int64

```
In [ . ]:
```

Acompanhando o retorno da função `.value_counts()` temos os números quanto aos agrupamentos. Esse dado nos será importante porque, podemos pegar o veículo que contém mais amostras, o mais comum deles, e usar como média para substituir os valores faltantes em outras categorias. A ideia é essa mesma, para não eliminarmos esses dados pode inconsistência, podemos preencher os espaços vazios com um valor médio.

```
22 base.loc[pd.isnull(base['vehicleType'])]
23 base['vehicleType'].value_counts()
24 base.loc[pd.isnull(base['gearbox'])]
25 base['gearbox'].value_counts()
26 base.loc[pd.isnull(base['model'])]
27 base['model'].value_counts()
28 base.loc[pd.isnull(base['fuelType'])]
29 base['fuelType'].value_counts()
30 base.loc[pd.isnull(base['notRepairedDamage'])]
31 base['notRepairedDamage'].value_counts()
```

Da mesma forma repetimos o processo para todas colunas onde houverem dados faltando (NaN), descobrindo os dados/valores médios para preencher tais espaços vazios.

```

33 valores_medios = {'vehicleType': 'limousine',
34                 'gearbox': 'manuell',
35                 'model': 'golf',
36                 'fuelType': 'benzin',
37                 'notRepairedDamage': 'nein'}

```

Em seguida criamos uma variável de nome valores_medios que recebe em forma de dicionário os parâmetros e seus respectivos dados/valores médios encontrados anteriormente.

```

39 base = base.fillna(value = valores_medios)
40

```

Uma vez identificados os dados faltantes em nossa base de dados, e descobertos os dados/valores médios para cada atributo previsor, hora de preencher esses espaços vários com dados. Aplicando sobre nossa variável base a função .fillna(), passando como parâmetro os dados do dicionário valores_medios, finalmente fazemos a devida alimentação e temos uma base de dados onde possamos operar sem problemas.

```

41 entradas = base.iloc[:, 1:13].values
42 saidas = base.iloc[:, 0].values

```

Com nossa base de dados íntegra, sem colunas desnecessárias, sem dados inconsistentes ou faltantes, podemos finalmente dividir tais dados em dados de entrada e de saída. Para isso criamos uma variável de nome entradas que recebe todos os valores de todas as linhas e todos valores das colunas 1 até a 13 por meio dos métodos .iloc[] e .values. Da mesma forma, criamos uma variável de nome saidas que recebe os valores de todas as linhas da coluna 0.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...
entradas	object	(359291, 12)	ndarray object of numpy module
saidas	object	(359291,)	ndarray object of numpy module
valores_medios	dict	5	{'vehicleType': 'limousine', 'gearbox': ...}
varTeste1	DataFrame	(12118, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...
varTeste2	DataFrame	(115, 15)	Column names: name, seller, offerType, price, abtest, vehicleType, yea ...

Se o processo ocorreu como esperado, podemos ver que de fato foram criadas as variáveis entradas e saidas. Entradas por sua vez tem um total de 12 colunas de atributos previsores e 359291 linhas de amostras a serem processadas. Note que estes números são bem inferiores aos da base em sua forma original, bruta. É absolutamente normal no processo de polimento eliminarmos boa parte da base de dados, em ciência de dados na verdade estamos mais preocupados com a integridade do que do número de dados a serem processados.

```

44 from sklearn.preprocessing import LabelEncoder
45

```

Como até o momento, mesmo após o polimento inicial, temos uma base de dados mista, contendo dados tanto em formato int (números) quanto string (texto), será necessário fazer a conversão desses dados todos para um tipo de dado de mesmo tipo. Para isso, da mesma forma como já feito no exemplo anterior, usaremos da ferramenta LabelEncoder para converter tudo em números a serem processados pela rede neural. O processo de importação se dá como de costume.

```

46 labelencoder = LabelEncoder()
47 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
48 entradas[:,1] = labelencoder.fit_transform(entradas[:,1])
49 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
50 entradas[:,5] = labelencoder.fit_transform(entradas[:,5])
51 entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
52 entradas[:,9] = labelencoder.fit_transform(entradas[:,9])
53 entradas[:,10] = labelencoder.fit_transform(entradas[:,10])

```

Como primeira parte desse processo de conversão, criamos nossa variável labelencoder que inicializa a ferramenta LabelEncoder, sem parâmetros mesmo. Na sequência sobre a coluna 0 de nossa base de dados aplicamos a função .fit_transform() que converte e atualiza os dados da coluna 0. O processo é feito para todas as colunas onde teremos de converter texto em número, ou seja, é replicado o processo também nas colunas 1, 2, 5, 8, 9 e 10.

```

55 from sklearn.preprocessing import OneHotEncoder
56

```

Como última parte desse processo de conversão, precisamos transformar esses dados em algo que seja indexado e interpretado pelo interpretador de nossa IDE corretamente. Tal processo é feito pela ferramenta OneHotEncoder. Importada como de costume do módulo preprocessing da biblioteca sklearn.

```

57 onehotencoder = OneHotEncoder(categorical_features = [0,1,3,5,8,9,10])
58 entradas = onehotencoder.fit_transform(entradas).toarray()

```

Em seguida criamos nossa variável de nome onehotencoder que recebe como atributo a ferramenta OneHotEncoder, que por sua vez recebe como parâmetro categorical_features = [0,1,3,5,8,9,10], em outras palavras, agora é feito a transformação de números brutos para valores categóricos indexados internamente. Logo após aplicamos sobre nossa variável entradas a função .fit_transform da ferramenta onehotencoder, convertendo tudo para uma array pela função .toarray().

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 12)	Column names: price, abtest, vehicleType, yearOfRegistration, gearbox, ...
entradas	float64	(359291, 316)	[0.00e+00 1.00e+00 0.00e+00 ... 0.00e+00 ...]
saidas	int64	(359291,)	[480 18300 9800 ... 9200 3400 289]
valores_medios	dict	5	{'vehicleType':'limousine', 'gearbox': ...}
varTeste1	DataFrame	(12118, 12)	Column names: price, abtest, vehicleType, yearOfRegistration, gearbox, ...
varTeste2	DataFrame	(115, 12)	Column names: price, abtest, vehicleType, yearOfRegistration, gearbox, ...

Note que após feitas as devidas conversões houve inclusive uma mudança estrutural na forma de apresentação dos dados, Os dados de entradas agora por sua vez possuem 359291 amostras categorizadas em 316 colunas. Não se preocupe, pois, essa conversão é internamente indexada para que se possam ser aplicadas funções sobre essa matriz.

entradas - Matriz NumPy

	0	1	2	3	
0	0	1	0	0	^
1	0	1	0	0	
2	0	1	0	0	
3	0	1	0	0	
4	0	1	0	0	
5	0	1	0	0	
6	0	1	0	0	
7	1	0	0	1	
8	0	1	0	0	
9	1	0	0	0	
10	1	0	0	0	
11	1	0	0	0	

< >

Formato Redimensionar Cor de fundo

Salvar e Fechar Fechar

Também é possível agora abrir a variável e visualmente reconhecer seu novo padrão.

Finalmente encerrada toda a fase de polimento de nossos dados, podemos dar início a próxima etapa que é a criação da estrutura de nossa rede neural artificial.

```
60 from keras.models import Sequential
61 from keras.layers import Dense
```

Como de praxe, todo processo se inicia com as importações das bibliotecas e módulos que serão usados em nosso código e que nativamente não são carregados nem pré-alocados em memória. Para este exemplo também estaremos criando uma rede neural densa (multicamada interconectada) e sequencial (onde cada camada é ligada à sua camada subsequente).

```

63 regressor = Sequential()
64 regressor.add(Dense(units = 158,
65                     activation = 'relu',
66                     input_dim = 316))
67 regressor.add(Dense(units = 158,
68                     activation = 'relu'))
69 regressor.add(Dense(units = 1,
70                     activation = 'linear'))
71 regressor.compile(loss = 'mean_absolute_error',
72                     optimizer = 'adam',
73                     metrics = ['mean_absolute_error'])

```

A estrutura se mantém muito parecida com as que construímos anteriormente, primeira grande diferença é que no lugar de um classificador estamos criando um regressor. Note que na primeira camada existem 316 neurônios, na segunda 158 e na última apenas 1. Outro ponto importante de salientar é que o processo de ativação da camada de saída agora é ‘linear’, este tipo de parâmetro de ativação é o mais comum de todos, uma vez que ele na verdade não aplica nenhuma função ao neurônio de saída, apenas pega o dado que consta nele e replica como valor final. No processo de compilação repare que tanto a função de perda quanto a métrica é definida como ‘mean_absolute_error’, em outras palavras, os pesos a serem corrigidos serão considerados de forma normal, absoluta, consultando a documentação do Keras você verá que outra função mito utilizada é a ‘squared_absolute_error’, onde o valor de erro é elevado ao quadrado e reaplicado na função como forma de penalizar mais os dados incorretos no processamento, ambos os métodos são válidos em um sistema de regressão, inclusive é interessante você realizar seus testes fazendo o uso dos dois.

```

74 regressor.fit(entradas,
75                 saídas,
76                 batch_size = 300,
77                 epochs = 100)

```

Estrutura criada, por meio da função .fit() alimentamos nossa rede com os dados de entradas, saídas, definimos que os pesos serão atualizados a cada 300 amostras e que a rede por sua vez será executada 100 vezes. Como sempre, é interessante testar taxas de atualização diferentes assim como executar a rede mais vezes a fim de que ela por aprendizado de reforço encontre melhores resultados.

```

Epoch 96/100
359291/359291 [=====] - 7s 18us/step - loss: 2238.8938
mean_absolute_error: 2238.8938
Epoch 97/100
359291/359291 [=====] - 7s 18us/step - loss: 2248.9770
mean_absolute_error: 2248.9770
Epoch 98/100
359291/359291 [=====] - 7s 18us/step - loss: 2237.1569
mean_absolute_error: 2237.1569
Epoch 99/100
359291/359291 [=====] - 7s 18us/step - loss: 2238.6745
mean_absolute_error: 2238.6745
Epoch 100/100
359291/359291 [=====] - 7s 18us/step - loss: 2242.1481
mean_absolute_error: 2242.1481
Out[ ]:

```

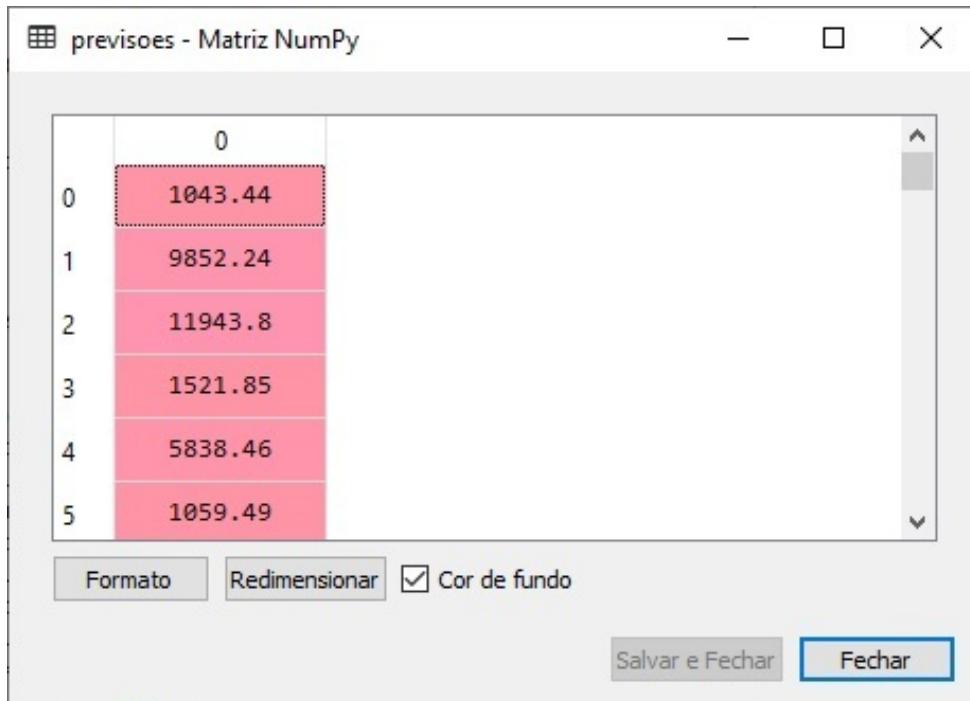
Terminado o processamento da rede note que o valor retornado foi de 2242.14, o que em outras palavras significa que a variação de valor desses carros em geral varia em 2242,14 euros para mais ou para menos.

```

79 previsoes = regressor.predict(entradas)
80 saidas.mean()
81 previsoes.mean()

```

Como de costume, sobre esses dados iniciais podemos aplicar nossos métodos preditivos. Para isso criamos uma variável de nome previsões que aplica a função `.predict()` em nosso regressor passando os dados contidos em entradas como parâmetro.



Explorando de forma visual a variável previsões podemos ver que de fato é feito um processamento interno usando outras funções aritméticas que resultam em diferentes valores para cada uma das amostras.

```

mean_absolute_error: 2238.6745
Epoch 100/100
359291/359291 [=====] - 7s 18us/step - loss: 2242.1481
mean_absolute_error: 2242.1481
Out[ ]: <keras.callbacks.History at 0x2534d0f8da0>

In [ ]: previsoes = regressor.predict(entradas)
...: saidas.mean()
...: previsoes.mean()
Out[ ]: 2357.3423

In [ ]: saidas.mean()
Out[ ]: 2916.833945186492

```

Via console podemos notar a proximidade dos resultados obtidos, o que nos indica que a rede está configurada corretamente. Uma grande disparidade entre esses valores pode indicar erros de processamento da rede.

```

83 from sklearn.model_selection import cross_val_score
84 from keras.wrappers.scikit_learn import KerasRegressor

```

Dando sequência podemos realizar sobre este modelo a mesma técnica de validação cruzada aplicada no modelo anterior. Como sempre, importamos o que é necessário, nesse caso a

ferramenta `cross_val_score` do módulo `model_selection` da biblioteca `sklearn`. Também estaremos aplicando e comparando resultados por meio do regressor que a biblioteca `keras` já tem dentro de si pré-configurado. Para isso importamos `KerasRegressor` do módulo `wrappers.scikit_learn` da biblioteca `keras`.

```
86 def regressorValCruzada():
87     regressorV = Sequential()
88     regressorV.add(Dense(units = 158,
89                           activation = 'relu',
90                           input_dim = 316))
91     regressorV.add(Dense(units = 158,
92                           activation = 'relu'))
93     regressorV.add(Dense(units = 1,
94                           activation = 'linear'))
95     regressorV.compile(loss = 'mean_absolute_error',
96                         optimizer = 'adam',
97                         metrics = ['mean_absolute_error'])
98
99     return regressorV
```

Prosseguindo, criamos nosso `regressorValCruzada` em forma de função. Internamente repare que a estrutura de rede neural é a mesma anterior. Usaremos essa função/rede para realizar novos testes.

```
100 regValCruzada = KerasRegressor(build_fn = regressorValCruzada,
101                                 epochs = 100,
102                                 batch_size = 300)
103 resValCruzada = cross_val_score(estimator = regValCruzada,
104                                   X = entradas,
105                                   y = saídas,
106                                   cv = 10,
107                                   scoring = 'neg_mean_absolute_error')
```

Para isso criamos nossa variável `regValCruzada` que recebe como atributo o `KerasRegressor`, que por sua vez tem como parâmetro `build_fn = regressorValCruzada`, o que em outras palavras ele inicializa nossa função criada antes internamente, executando a rede 100 vezes, atualizando os pesos a cada 300 amostras. Na sequência, criamos nossa variável de nome `resValCruzada`, que irá executar a validação cruzada propriamente dita, então, por sua vez, `resValCruzada` recebe como atributo `cross_val_score` que tem como parâmetros `estimator = regValCruzada`, ou seja, aplica seu processamento sobre os dados obtidos de nossa função `regressorValCruzada`, usando `X` e `y` como dados de parâmetro, dados a serem usados diretamente de nossa base para comparação, na sequência é definido que `cv = 10`, em outras palavras, toda base será dividida em 10 partes iguais e testadas individualmente, por fim `scoring = 'neg_mean_absolute_error'` fará a apresentação da média dos resultados obtidos, desconsiderando o sinal dos mesmos, apenas os números em seu estado bruto.

```
Epoch 96/100
323361/323361 [=====] - 7s 21us/step - loss: 2266.0997
mean_absolute_error: 2266.0997
Epoch 97/100
323361/323361 [=====] - 8s 26us/step - loss: 2254.7366
mean_absolute_error: 2254.7366
Epoch 98/100
323361/323361 [=====] - 8s 24us/step - loss: 2248.4811
mean_absolute_error: 2248.4811
Epoch 99/100
323361/323361 [=====] - 8s 24us/step - loss: 2250.4997
mean_absolute_error: 2250.4997
Epoch 100/100
323361/323361 [=====] - 6s 19us/step - loss: 2253.2591
mean_absolute_error: 2253.2591
```

Selecionando e executando todo bloco de código é possível acompanhar via console a rede neural sendo executada, lembrando que esta etapa costuma ser bastante demorada uma vez que toda rede será executada uma vez para treino e mais 10 vezes para testar cada parte da base.



Terminado o processo de validação cruzada é possível ver os valores obtidos para cada uma das partes da base de dados testada individualmente.

```
108 media = resValCruzada.mean()  
109 desvioPadrao = resValCruzada.std()
```

Finalizando nosso exemplo, criamos as variáveis dedicadas a apresentar os resultados finais de nossa validação cruzada para as devidas comparações com os dados de nossa rede neural artificial.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(359291, 12)	Column names: price, abtest, vehicleType, ... 51.95240597776519
desvioPadrao	float64	1	51.95240597776519
entradas	float64	(359291, 316)	[[0.00e+00 1.00e+00 0.00e+00 ... 0.00e+00] [0.00e+0 ...]
media	float64	1	-2238.8809286392097
previsoes	float32	(359291, 1)	[[1043.4368] [9852.238]
resValCruzada	float64	(10,)	[-2283.76042768 -2218.33065824 -2334.1188 ... -2175.92 ...]
saidas	int64	(359291,)	[480 18300 9800 ... 9200 3400 289901]

E o importante a destacar é que, como nos exemplos anteriores, estes testes são realizados como um viés de confirmação para termos certeza da integridade de nossos dados, aqui, realizados os devidos testes, podemos concluir que os resultados são próximos, confirmado que a execução de nossa rede se deu de forma correta.

Código Completo:

```

1 import pandas as pd
2 from sklearn.preprocessing import LabelEncoder
3 from sklearn.preprocessing import OneHotEncoder
4 from keras.models import Sequential
5 from keras.layers import Dense
6 from sklearn.model_selection import cross_val_score
7 from keras.wrappers.scikit_learn import KerasRegressor
8
9 base = pd.read_csv('autos.csv', encoding = 'ISO-8859-1')
10
11 base = base.drop('dateCrawled', axis=1)
12 base = base.drop('dateCreated', axis=1)
13 base = base.drop('nrOfPictures', axis=1)
14 base = base.drop('postalCode', axis=1)
15 base = base.drop('lastSeen', axis=1)
16 base = base.drop('name', axis=1)
17 base = base.drop('seller', axis=1)
18 base = base.drop('offerType', axis=1)
19
20 varTeste1 = base.loc[base.price <= 10]
21 base = base[base.price > 10]
22 varTeste2 = base.loc[base.price > 350000]
23 base = base[base.price < 350000]
```

```

25 base.loc[pd.isnull(base['vehicleType'])]
26 base['vehicleType'].value_counts()
27 base.loc[pd.isnull(base['gearbox'])]
28 base['gearbox'].value_counts()
29 base.loc[pd.isnull(base['model'])]
30 base['model'].value_counts()
31 base.loc[pd.isnull(base['fuelType'])]
32 base['fuelType'].value_counts()
33 base.loc[pd.isnull(base['notRepairedDamage'])]
34 base['notRepairedDamage'].value_counts()
35
36 valores_medios = {'vehicleType': 'limousine',
37                     'gearbox': 'manuell',
38                     'model': 'golf',
39                     'fuelType': 'benzin',
40                     'notRepairedDamage': 'nein'}
41
42 base = base.fillna(value = valores_medios)
43 entradas = base.iloc[:, 1:13].values
44 saidas = base.iloc[:, 0].values
45
46 labelencoder = LabelEncoder()
47 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
48 entradas[:,1] = labelencoder.fit_transform(entradas[:,1])
49 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
50 entradas[:,5] = labelencoder.fit_transform(entradas[:,5])
51 entradas[:,8] = labelencoder.fit_transform(entradas[:,8]) #
52 entradas[:,9] = labelencoder.fit_transform(entradas[:,9])
53 entradas[:,10] = labelencoder.fit_transform(entradas[:,10])
54
55 onehotencoder = OneHotEncoder(categorical_features = [0,1,3,5,8,9,10])
56 entradas = onehotencoder.fit_transform(entradas).toarray()
57
58 regressor = Sequential()
59 regressor.add(Dense(units = 158,
60                     activation = 'relu',
61                     input_dim = 316))
62 regressor.add(Dense(units = 158,
63                     activation = 'relu'))
64 regressor.add(Dense(units = 1,
65                     activation = 'linear'))
66 regressor.compile(loss = 'mean_absolute_error',
67                     optimizer = 'adam',
68                     metrics = ['mean_absolute_error'])
69 regressor.fit(entradas,
70                 saidas,
71                 batch_size = 300,
72                 epochs = 100)
73
74 previsoes = regressor.predict(entradas)
75 saidas.mean()
76 previsoes.mean()

```

```
78 def regressorValCruzada():
79     regressorV = Sequential()
80     regressorV.add(Dense(units = 158,
81                         activation = 'relu',
82                         input_dim = 316))
83     regressorV.add(Dense(units = 158,
84                         activation = 'relu'))
85     regressorV.add(Dense(units = 1,
86                         activation = 'linear'))
87     regressorV.compile(loss = 'mean_absolute_error',
88                         optimizer = 'adam',
89                         metrics = ['mean_absolute_error'])
90     return regressorV
91
92 regValCruzada = KerasRegressor(build_fn = regressorValCruzada,
93                                 epochs = 100,
94                                 batch_size = 300)
95 resValCruzada = cross_val_score(estimator = regValCruzada,
96                                 X = entradas,
97                                 y = saidas,
98                                 cv = 10,
99                                 scoring = 'neg_mean_absolute_error')
100 media = resValCruzada.mean()
101 desvioPadrao = resValCruzada.std()
```

Regressão com Múltiplas Saídas – VGDB Dataset

Uma das aplicações bastante comuns em ciência de dados é tentarmos estipular números de vendas de um determinado produto, para a partir disso gerar algum prospecto de mudanças em seu modelo de vendas. Aqui, para entendermos esse tipo de conceito de forma prática, usaremos uma base de dados de vendas de jogos de videogame onde temos as informações de suas vendas em três mercados diferentes. Temos dados brutos das vendas dos jogos na américa do norte, na europa e no japão, após o devido polimento e processamento dos dados dessa base, poderemos ver qual região obteve maior lucro em suas vendas para assim planejar estratégias de vendas nas outras de menor lucro. Para isto estaremos usando uma base de dados real disponível gratuitamente no repositório Kaggle.

A screenshot of a Kaggle dataset page. The title is "Video Game Sales" with a subtitle "Analyze sales data from more than 16,500 games." Below the title is a photo of a person holding a game controller. The author is listed as "GregorySmith · updated 3 years ago (Version 2)". There are tabs for "Data", "Kernels (331)", "Discussion (13)", "Activity", and "Metadata". A "Download (403 KB)" button is available, along with a "New Notebook" button. Below the tabs, there are filters for "Usability 7.6" and "Tags video games". A "Description" section contains the text: "This dataset contains a list of video games with sales greater than 100,000 copies. It was generated by a scrape of vgchartz.com." It also lists "Fields include": "Rank - Ranking of overall sales" and "Name - The games name".

Dentro do site do Kaggle, basta procurar por Video Game Sales e da página do dataset fazer o download do respectivo arquivo em formato .csv.

Partindo para o código:

A screenshot of a code editor window titled "VGDB.py". The code starts with three imports: "import pandas as pd", "from keras.layers import Dense, Input", and "from keras.models import Model".

Inicialmente criamos um novo arquivo Python 3 de nome VGDB.py, em seguida, como sempre fazemos, realizamos as devidas importações das bibliotecas, módulos e ferramentas que nos auxiliarão ao longo do código. Repare que aqui temos uma pequena diferença em relação aos exemplos anteriores, não importamos Sequential (módulo que criava e linkava automaticamente as camadas de nossa rede neural) e importamos o módulo Model, que por sua vez nos permitirá criar uma rede neural densa com múltiplas saídas.

```
5 base = pd.read_csv('games.csv')
6
```

Em seguida criamos uma variável de nome base que recebe por meio da função `.read_csv` o conteúdo de nosso arquivo `games.csv` parametrizado aqui.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(16719, 16)	Column names: Name, Platform, Year_of_Release, Ger Publisher, NA_Sa ...

Uma vez realizada a importação é possível via explorador de variáveis ver que de fato foi criada nossa variável base em forma de DataFrame, onde inicialmente temos 16719 objetos/amostras/registros com atributos divididos em 16 categorias diferentes.

base - DataFrame

Index	Name	Platform	Year_of_Release	Genre	Publisher
0	Wii Sports	Wii	2006	Sports	Nintendo
1	Super Mario Bros.	NES	1985	Platform	Nintendo
2	Mario Kart Wii	Wii	2008	Racing	Nintendo
3	Wii Sports Resort	Wii	2009	Sports	Nintendo
4	Pokemon Red/Pokemon Blue	GB	1996	Role-Playing	Nintendo
5	Tetris	GB	1989	Puzzle	Nintendo
6	New Super Mario Bros.	DS	2006	Platform	Nintendo
7	Wii Play	Wii	2006	Misc	Nintendo
8	New Super Mario Bros. ...	Wii	2009	Platform	Nintendo
9	Duck Hunt	NES	1984	Shooter	Nintendo
10	Nintendogs	DS	2005	Simulation	Nintendo
11	Mario Kart DS	DS	2005	Racing	Nintendo
12	Pokemon Gold/Pokemon Silv...	GB	1999	Role-Playing	Nintendo
13	Wii Fit	Wii	2007	Sports	Nintendo

Formato Cor de fundo Min/max de

Analizando o conteúdo de base, podemos ver que, como de costume, temos muitas inconsistências nos dados, muitos dados faltando e por fim, colunas de informação desnecessárias para nosso propósito. Sendo assim iniciamos a fase de polimento dos dados removendo as colunas desnecessárias.

```
7 base = base.drop('Other_Sales', axis = 1)
8 base = base.drop('Global_Sales', axis = 1)
9 base = base.drop('Developer', axis = 1)
10 base = base.dropna(axis = 0)
11 base = base.loc[base['NA_Sales'] > 1]
12 base = base.loc[base['EU_Sales'] > 1]
```

Diretamente sobre nossa variável base aplicaremos algumas funções. Primeiramente por meio da função .drop() eliminamos as colunas Other_Sales, Global_Sales e Developer, note que para eliminar colunas de nossa base devemos colocar como parâmetro também axis = 1. Na sequência por meio da função .dropna() excluímos todas as linhas onde haviam valores faltando (não zerados, faltando mesmo), e agora como estamos aplicando função sobre linhas, agora axis = 0.

Por fim, por meio da expressão `.loc[base['NA_Sales'] > 1]` mantemos dessa coluna apenas as linhas com valores maiores que 1, o mesmo é feito para EU_Sales.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(258, 13)	Column names: Name, Platform, Year_of_Release, Genre, Publisher, NA_Sales ...

Repare que após a aplicação das funções anteriores, houve uma redução considerável no número de amostras que temos. O que indica que dentro dos parâmetros que definimos haviam muitas inconsistências de dados, e isto é perfeitamente normal quando pegarmos bases de dados geradas tanto por sistemas quanto por usuários, o importante é que dos poucos dados que sobrem, estes sejam íntegros para que possamos processá-los e extrair informações a partir dos mesmos.

```
14 base['Name'].value_counts()
15 backupName = base.Name
16 base = base.drop('Name', axis = 1)
```

Dando sequência realizaremos um procedimento de extrair os dados de uma coluna inteira de nossa base, porém não iremos descartar tais dados, apenas separá-los da base salvando-os em uma nova variável. Para isso aplicando sobre a coluna Name de nossa base a função `.value_counts()` fazemos a leitura de todos os dados/valores que houverem nessa coluna. Em seguida criamos uma variável de nome backupBase que recebe os dados contidos em `base.Names`. Por fim, por meio da função `.drop()` eliminamos de nossa base a coluna inteira Name.

```
18 entradas = base.iloc[:, [0,1,2,3,7,8,9,10,11]].values
19 vendas_NA = base.iloc[:, 4].values
20 vendas_EU = base.iloc[:, 5].values
21 vendas_JP = base.iloc[:, 6].values
```

Eliminadas as inconsistências de nossa base de dados, hora de dividir a mesma em dados de entrada e de saída. Inicialmente criamos uma variável de nome entradas que recebe como atributo todos os dados das colunas 0, 1, 2, 3, 7, 8, 9, 10 e 11 por meio de `.iloc[].values`. Na sequência criamos 3 variáveis de saída, uma vez que neste exemplo é isto que queremos. A variável `vendas_NA` recebe os dados de todas as linhas da coluna 4, `vendas_EU` recebe o conteúdo de todas as linhas da coluna 5 e por fim `vendas_JP` recebe todas as linhas da coluna 6.

Nome	Tipo	Tamanho	Valor
backupName	Series	(258,)	Series object of pandas.core.series module
base	DataFrame	(258, 12)	Column names: Platform, Year_of_Release, Genre, Publisher, NA_Sales, EU_Sales, JP_Sales ...
entradas	object	(258, 9)	ndarray object of numpy module
vendas_EU	float64	(258,)	[28.96 12.76 10.93 ... 1.05 1.12 1.19]
vendas_JP	float64	(258,)	[3.77 3.79 3.28 ... 0.24 0.06 0.]
vendas_NA	float64	(258,)	[41.36 15.68 15.61 ... 1.05 1.13 1.07]

Se não houver nenhum erro de sintaxe nesse processo podemos ver que as devidas variáveis foram criadas. Note que temos dados numéricos e em forma de texto, sendo assim, próxima etapa da fase de polimento é converter todos dados para tipo numérico para que possam ser processados pela rede.

```
23 from sklearn.preprocessing import LabelEncoder, OneHotEncoder
24
```

Como sempre, realizamos as devidas importações das ferramentas que inicialmente não são pré-carregadas por nossa IDE.

```
25 labelencoder = LabelEncoder()
26 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
27 entradas[:,2] = labelencoder.fit_transform(entradas[:,2])
28 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
29 entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
```

Dando sequência criamos nossa variável labelencoder que inicializa a ferramenta LabelEncoder, sem parâmetros mesmo. Na sequência é aplicada a função .fit_transform() sobre as colunas 0, 2, 3 e 8, transformando seus dados do tipo string para int.

```
31 onehotencoder = OneHotEncoder(categorical_features = [0,2,3,8])
32 entradas = onehotencoder.fit_transform(entradas).toarray()
```

Próximo passo é tornar esse conteúdo convertido indexado de forma que nosso interpretador consiga fazer a correta leitura dos dados e aplicar funções sobre os mesmos. Logo, criamos nossa variável de nome onehotencoder que executa a ferramenta OneHotEncoder e seu parâmetro categorical_features sobre as colunas 0, 2, 3 e 8. Por fim, nossa variável entradas recebe a conversão de tais dados para uma array do tipo numpy por meio das funções .fit_transform().toarray().

```
34 camada_entrada = Input(shape = (61, ))
35 camada_oculta1 = Dense(units = 32,
                           activation = 'sigmoid')(camada_entrada)
36 camada_oculta2 = Dense(units = 32,
                           activation = 'sigmoid')(camada_oculta1)
37 camada_saida1 = Dense(units = 1,
                        activation = 'linear')(camada_oculta2)
38 camada_saida2 = Dense(units = 1,
                        activation = 'linear')(camada_oculta2)
39 camada_saida3 = Dense(units = 1,
                        activation = 'linear')(camada_oculta2)
40
41
42
43
44
```

Finalmente terminada a fase de polimento dos dados, damos início a criação da estrutura de nossa rede neural artificial. Nos modelos anteriores estivemos criando redes neurais sequenciais. Aqui criaremos um modelo um pouco diferente, ainda sequencial mas iremos criar os laços entre as camadas de forma totalmente manual, por fim, a segunda grande diferença é que estaremos criando um modelo de rede que processará uma base de dados para gerar múltiplas saídas independentes, uma vez que aqui estamos trabalhando para descobrir preços de vendas em 3 locações diferentes.

Inicialmente criamos nossa variável camada_entrada que recebe como atributo Input que por sua vez recebe o formato da array gerada anteriormente, note que não estamos pegando as amostras e transformando em neurônios da camada de entrada, as amostras por sua vez foram convertidas e indexadas como parte de uma matriz. Em seguida criamos uma variável de nome camada_oculta1 que recebe como atributo Dense que por sua vez tem como parâmetros units = 32 e activation = 'sigmoid', em outras palavras, nossa rede que tem 61 neurônios em sua camada de entrada, agora tem 32 em sua primeira camada oculta, e o processamento desses neurônios com seus respectivos pesos deve gerar uma probabilidade entre 0 e 1 por causa da função de ativação escolhida ser a sigmoide. Por fim, para haver a comunicação entre uma camada e outra a referência da camada é passada como um segundo parâmetro independente. Para finalizar são criadas 3 camadas de saída, cada uma com 1 neurônio, activation = linear (ou seja, não é aplicada nenhuma função, apenas replicado o valor encontrado) e as devidas linkagens com camada_oculta2.

```

46 regressor = Model(inputs = camada_entrada,
47                     outputs = [camada_saida1,
48                               camada_saida2,
49                               camada_saida3])
50 regressor.compile(optimizer = 'adam',
51                     loss = 'mse')
52 regressor.fit(entradas,
53                 [vendas_NA, vendas_EU, vendas_JP],
54                 epochs = 5000,
55                 batch_size = 100)

```

Em seguida é criado nosso regressor, por meio da variável homônima, que recebe como atributo Model que por sua vez tem como parâmetros inputs = camada_entrada e outputs = [camada_saida1, camada_saida2, camada_saida3]. Logo após é executado sobre nosso regressor a função .compile que define o modo de compilação do mesmo, optimizer = ‘adam’ e loss = ‘mse’. Adam já é um otimizador conhecido nosso de outros modelos, mas neste caso ‘mse’ significa mean squared error, é um modelo de função de perda onde o resultado encontrado é elevado ao quadrado e reaplicado na rede, é uma forma de dar mais peso aos erros e forçar a rede a encontrar melhores parâmetros de processamento. Por fim alimentamos nosso regressor por meio da função .fit() com os respectivos dados de entrada e de saída, assim como definimos que a rede será executada 5000 vezes, atualizando os pesos a cada 100 amostras.

```

57 #previsor
58 previsao_NA, previsao_EU, previsao_JP = regressor.predict(entradas)
59

```

Como de costume, é interessante realizar previsões e as devidas comparações para testarmos a eficiência de nossa rede. Para isso criamos as variáveis previsao_NA, previsao_EU e previsao_JP que recebe nosso regressor, aplicando a função .predict() diretamente sobre nossos dados de entradas.

backupName - Series		previsao_NA - Matriz N		vendas_NA - Matriz NumPy	
Index	Name		0		0
0	Wii Sports	0	41.1149	0	41.36
2	Mario Kart	1	21.3701	1	15.68
	Wii	2	15.6081	2	15.61
3	Wii Sports	3	16.6921	3	11.28
	Resort	4	10.7007	4	13.96
6	New Super	5	13.0384	5	14.44
	Mario Bros.	6	15.5568	6	9.71
7	Wii Play	7	12.1215	7	8.92
8	New Super	8	8.7159	8	15
	Mario Bros. ...	9	11.2487	9	9.01
11	Mario Kart DS	10	4.89999	10	7.02
13	Wii Fit	11	6.74438	11	4.74
14	Kinect Adventures!	12	4.9	12	9.66
15	Wii Fit Plus				
16	Grand Theft Auto V				
19	Brain Age: Train Your Br...				
23	Grand Theft Auto V				
24	Grand Theft Auto: Vice C...				

Cor de fundo

Por fim podemos equiparar e comparar as colunas dos respectivos nomes dos jogos, os valores reais extraídos de nossa base de dados e os valores encontrados por nossa rede neural artificial, note que de fato os dados são próximos, ou pelo menos possuem uma margem de erro dentro do aceitável. Sendo assim, podemos salvar esse modelo para reutilizar quando for necessário resolver algum tipo de problema parecido.

Código Completo:

```

1 import pandas as pd
2 from keras.layers import Dense, Input
3 from keras.models import Model
4 from sklearn.preprocessing import LabelEncoder, OneHotEncoder
5
6 base = pd.read_csv('games.csv')
7 base = base.drop('Other_Sales', axis = 1)
8 base = base.drop('Global_Sales', axis = 1)
9 base = base.drop('Developer', axis = 1)
10 base = base.dropna(axis = 0)
11 base = base.loc[base['NA_Sales'] > 1]
12 base = base.loc[base['EU_Sales'] > 1]
13 base['Name'].value_counts()
14 backupName = base.Name
15 base = base.drop('Name', axis = 1)
16
17 entradas = base.iloc[:, [0,1,2,3,7,8,9,10,11]].values
18 vendas_NA = base.iloc[:, 4].values
19 vendas_EU = base.iloc[:, 5].values
20 vendas_JP = base.iloc[:, 6].values
21
22 labelencoder = LabelEncoder()
23 entradas[:,0] = labelencoder.fit_transform(entradas[:,0])
24 entradas[:,2] = labelencoder.fit_transform(entradas[:,2])
25 entradas[:,3] = labelencoder.fit_transform(entradas[:,3])
26 entradas[:,8] = labelencoder.fit_transform(entradas[:,8])
27 onehotencoder = OneHotEncoder(categorical_features = [0,2,3,8])
28 entradas = onehotencoder.fit_transform(entradas).toarray()
29
30 camada_entrada = Input(shape = (61, ))
31 camada_oculta1 = Dense(units = 32,
32                         activation = 'sigmoid')(camada_entrada)
33 camada_oculta2 = Dense(units = 32,
34                         activation = 'sigmoid')(camada_oculta1)
35 camada_saida1 = Dense(units = 1,
36                         activation = 'linear')(camada_oculta2)
37 camada_saida2 = Dense(units = 1,
38                         activation = 'linear')(camada_oculta2)
39 camada_saida3 = Dense(units = 1,
40                         activation = 'linear')(camada_oculta2)
41
42 regressor = Model(inputs = camada_entrada,
43                     outputs = [camada_saida1,
44                                camada_saida2,
45                                camada_saida3])
46 regressor.compile(optimizer = 'adam',
47                     loss = 'mse')
48 regressor.fit(entradas,
49                 [vendas_NA, vendas_EU, vendas_JP],
50                 epochs = 5000,
51                 batch_size = 100)
52 #previsor
53 previsao_NA, previsao_EU, previsao_JP = regressor.predict(entradas)

```

Previsão Quantitativa – Publi Dataset

Uma das aplicações de ciência de dados bastante recorrente é a de identificar algum retorno (em valores monetários) com base no investimento feito. Em outras palavras é possível identificar via processamento de aprendizado de máquina, se o investimento em um determinado nicho surtiu de fato algum resultado. Aqui estaremos usando uma database que com base no investimento em algumas categorias de publicidade de um determinado produto, possamos ver onde esse investimento surtiu melhor efeito, levando em consideração qual foi o retorno de acordo com cada unidade monetária investida.

An Introduction to Statistical Learning

with Applications in R

Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani

[Home](#)

[About this Book](#)

[R Code for Labs](#)

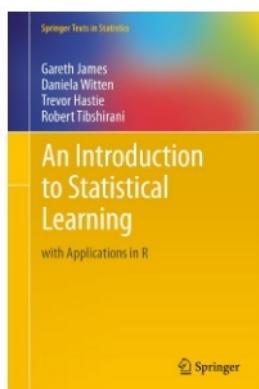
[Data Sets and Figures](#)

[ISLR Package](#)

[Get the Book](#)

[Author Bios](#)

[Errata](#)



[Figures](#)

[Data Sets](#)

[All Figures In A Single Zip File](#)

[Advertising.csv](#)

A base de dados que usaremos dessa vez não está nem no UCI nem no Kaggle, porém você pode ter acesso a essa base de dados por meio do site acima. Clicando sobre Advertising.csv você fará o download do referente arquivo.

```
Advertising Dataset.py
1 import pandas as pd
2
```

Como sempre, inicialmente criamos um arquivo Python 3 para nosso código e realizamos as devidas importações iniciais.

```
3 base = pd.read_csv('Advertising.csv')
4 print(base.head())
```

Em seguida criamos nossa variável de nome base que irá receber todo o conteúdo de Advertising.csv por meio da função .read_csv() do pandas. A partir desse momento se o processo de importação ocorreu sem erros, a variável base será devidamente criada. Uma das maneiras de visualizar rapidamente o seu conteúdo (principalmente se você estiver usando outra IDE diferente do Spyder) é executar o comando print() parametrizado com base.head(), dessa forma será exibida em console as 5 primeiras linhas desse dataframe.

```

Console 1/A

In [ ]: import pandas as pd
...
...: base = pd.read_csv('Advertising.csv')

In [ ]: print(base.head())
      Unnamed: 0    TV  radio  newspaper  sales
0            1  230.1   37.8       69.2  22.1
1            2   44.5   39.3       45.1  10.4
2            3   17.2   45.9       69.3   9.3
3            4  151.5   41.3       58.5  18.5
4            5  180.8   10.8       58.4  12.9

```

Note que existem 6 colunas, sendo duas delas indexadores, um da própria planilha Excel e outro gerado pelo pandas durante a importação. Essas colunas simplesmente não serão utilizadas como referência nesse modelo. Também temos 3 colunas com atributos previsores (TV, radio e newspaper) e uma coluna com dados de saída (sales). O que iremos fazer nesse modelo é executar algumas funções interessantes fora de uma rede neural densa, apenas para também visualizar estas possibilidades, uma vez que dependendo muito da complexidade do problema o mesmo não precisa ser processado por uma rede neural.

```

9 import seaborn as sns
10

```

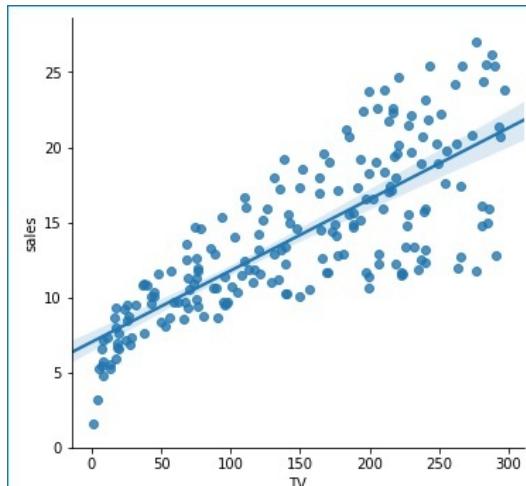
Dando sequência realizamos a importação da biblioteca Seaborn, muito utilizada em alternativa a Matplotlib, de finalidade parecida, ou seja, exibir os dados de forma visual em forma de gráficos.

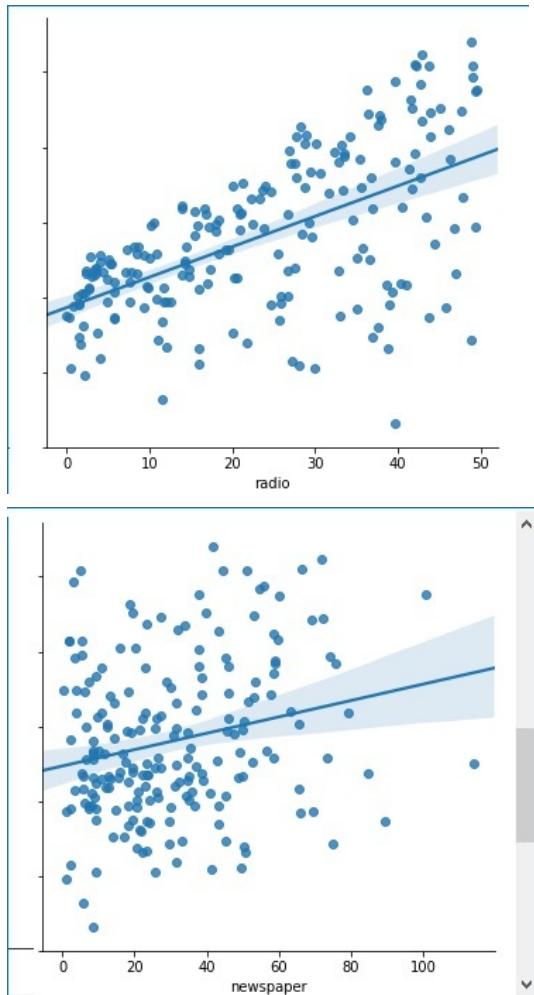
```

11 sns = sns.pairplot(base,
12                      x_vars = ['TV', 'radio', 'newspaper'],
13                      y_vars = 'sales',
14                      size = 5,
15                      kind = 'reg')

```

Em seguida criamos nossa variável sns, que por sua vez recebe como atributo a função sns.pairplot(), onde passamos como parâmetros todo conteúdo de base, x_vars recebe os atributos previsores (os que serão plotados no eixo X), y_vars que recebe os dados de saída de sales, size = 5 simplesmente para visualização mais clara dos gráficos e por fim kind = 'reg', que define que o tipo de apresentação dos dados no gráfico é o tipo de dados de regressão.





Analisando rapidamente os gráficos podemos encontrar a relação entre cada atributo previsões e o valor de vendas, importante salientar que aqui por hora, o que existe de mais relevante nesses gráficos é a angulação da linha gerada, uma linha ascendente confirma que houve retorno positivo de acordo com o investimento, pode acontecer de existir linhas descendentes em um ou mais gráficos de amostras, representando prejuízo em relação ao investimento. Nesse exemplo, ambos os 3 previsores (veículos de mídia) retornaram estimativas positivas, de lucro com base em seus dados.

```

17 from sklearn.model_selection import train_test_split
18
19 etreino, eteste, streino, steste = train_test_split(entradas,
20                                     saídas,
21                                     test_size = 0.3)

```

Tendo as primeiras estimativas de resultado, hora de aplicarmos alguma testes para avaliar a performance de nosso modelo. Como de costume, o processo se inicia importando o que for necessário, aqui, usaremos inicialmente a ferramenta `train_test_split`. Logo após criamos as variáveis dedicadas a treino e teste do modelo, bem como a definição de que 30% das amostras serão reservadas para teste, sobrando 70% para treino.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, s
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
etreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module
streino	Series	(140,)	Series object of pandas.core.series module

Dando uma conferida no nosso explorador de variáveis podemos ver que de fato foram criadas tais variáveis.

```
23 from sklearn.linear_model import LinearRegression
24
```

Em seguida realizamos a importação da ferramenta LinearRegression, do módulo linear_model, da biblioteca sklearn.

```
25 reglinear = LinearRegression()
26 reglinear.fit(etreino,streino)
```

A seguir criamos nossa variável de nome reglinear que inicialmente apenas inicializa a ferramenta LinearRegression, sem parâmetros mesmo. Depois aplicamos sobre reglinear a função .fit() passando como parâmetro os dados de etreino e streino.

```
28 print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
29
```

Uma vez executado o modelo de regressão linear podemos agora aplicar alguns testes. Primeiramente instanciamos o conteúdo de nossos atributos previsores, seguido do argumento reglinear.coef_ que em outras palavras, retornará o valor de coeficiente (de investimento nesse caso).

```
In [8]: print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
[('TV', 0.0473674298724549), ('radio', 0.1853722490601593), ('newspaper',
 -0.00238463121485491)]
```

Selecionando e executando o bloco de código anterior teremos um retorno padrão via console, onde podemos ver que existem coeficientes para cada um de nossos atributos previsores, estes coeficientes para entendimento mais fácil podem ser interpretados como valores de unidade monetária, por exemplo em ‘TV’ temos 0.04, o que significa que para cada 1 dólar investido, houve um aumento nas vendas de 4%, seguindo o mesmo raciocínio lógico, em ‘radio’ para cada dólar investido houve um aumento nas vendas de 18% e por fim em ‘newspaper’ para cada dólar investido houve um valor nulo, você até poderia em outras situações considerar 0.002 centavos de dólar de prejuízo, aqui nosso parâmetro é de apenas duas casas decimais mesmo, sendo que para cada 1 dólar investido não houve nem lucro nem prejuízo nas vendas.

```
30 print(reglinear.predict([[230.1,37.8,69.2]]))
31
```

Por meio da função .predict() podemos passar em forma de lista uma das linhas de nossa base de dados, contendo os três atributos previsores e realizar teste sobre essa amostra.

```
In [ ]: print(reglinear.predict([[230.1,37.8,69.2]]))  
[20.64299359]
```

Usando esses parâmetros como exemplo, numa campanha onde foi investido 230.1 dólares em publicidade via TV, 37.8 dólares em publicidade via rádio e 69.2 dólares em publicidade via jornal, houve um aumento nas vendas em geral de 20.6%. Repare que o interessante desse teste em particular é justamente equiparar quanto foi investido para cada mídia e qual foi o retorno, para justamente realizar ajustes e focar nas próximas campanhas na mídia que deu maior retorno.

```
32 previsor = reglinear.predict(eteste)  
33 print(previsor)
```

Como já fizemos algumas vezes em outros modelos, podemos criar nossa variável de nome previsor que aplica a função .predict() sobre toda a nossa base separada em eteste.

```
In [ ]: previsor = reglinear.predict(eteste)  
...: print(previsor)  
[21.99131061 10.37845701 15.52067254 14.98257988 7.67663135 12.18712659  
17.38487641 17.41116216 8.67769227 6.54954127 12.08501628 12.55749281  
20.92556622 11.16525127 14.45488414 9.75249818 13.91969088 8.3366973  
10.0741478 16.3781017 10.36147754 9.49902928 11.32164028 15.50715257  
21.93928981 8.15167538 19.28286002 15.46912896 9.42837231 15.95689331  
4.40108511 6.05415038 17.91265432 16.92633817 9.87808944 21.33738647  
13.61495178 9.69074923 5.75556433 17.04488883 14.45251759 19.1892124  
15.89857622 8.94867862 14.30579785 8.83091711 7.71232706 14.07316039  
15.07185971 18.57963053 15.29149013 10.18037182 8.35771214 21.243932  
7.37650581 18.68617391 16.36841081 11.64848861 7.89186185 12.82134607]
```

Selecionado e executado o bloco de código anterior, os resultados são exibidos em console, assim como atribuídos a nossa variável previsor.

The image displays two windows side-by-side. The left window is titled 'steste - DataFrame' and contains a table with four columns: 'Index', 'TV', 'radio', and 'newspap'. The right window is titled 'previsor - Matriz NumPy' and contains a vertical list of 13 numerical values. Both windows have standard window controls (minimize, maximize, close) at the top right.

Index	TV	radio	newspap
58	210.8	49.6	37.7
12	23.8	35.1	65.9
87	110.7	40.6	63.2
110	225.8	8.2	56.5
34	95.7	1.4	7.4
1	44.5	39.3	45.1
41	177	33.4	38.7
35	290.7	4.1	8.5
78	5.4	29.9	9.4
182	56.2	5.7	29.7
116	139.2	14.3	25.6
179	165.6	10	17.6
185	205	45.1	19.6
164	117.2	14.7	5.4

0
21.9913
10.3785
15.5207
14.9826
7.67663
12.1871
17.3849
17.4112
8.67769
6.54954
12.085
12.5575
20.9256

Buttons at the bottom of both windows include 'Formato', 'Redimensionar', 'Salvar e Fechar', and 'Fechar'.

Abrindo as devidas variáveis via explorador de variáveis podemos fazer uma fácil relação, onde dentro de um período, os valores investidos em campanhas publicitárias surtiram um certo percentual de aumento nas vendas em geral.

```
35 from sklearn import metrics
36
```

Podemos ainda realizar alguns testes para confirmar a integridade de nossos dados. Aqui usaremos da ferramenta metrics da biblioteca sklearn para tal fim.

```
37 mae = metrics.mean_absolute_error(steste,previsor)
38
```

Primeiro teste será realizado por meio da função `.mean_absolute_error()` que com base nos dados de steste e dos obtidos em previsor, gerará um valor atribuído a nossa variável mae.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
estreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...]
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module
streino	Series	(140,)	Series object of pandas.core.series module

Se não houve nenhum erro de sintaxe até o momento será criada a variável mae, note que ela possui o valor 1.41, que em outras palavras significa que para cada 1 dólar investido, houve um retorno de \$1.41.

```
39 mse = metrics.mean_squared_error(steste,previsor)
40
```

Em seguida nos mesmos moldes aplicamos a função .mean_squared_error() que por sua vez fará o processamento dos dados sobre steste e previsor, porém com a particularidade que aqui os valores de erro encontrados durante o processamento são elevados ao quadrado, dando mais peso aos erros. Raciocine que por exemplo um erro 2 elevado ao quadrado é apenas 4, enquanto um erro 5 ao quadrado é 25, o que gera grande impacto sobre as funções aplicadas.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
estreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
mse	float64	1	3.2825652471496274
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...]
saidas	Series	(200,)	Series object of pandas.core.series module
steste	Series	(60,)	Series object of pandas.core.series module

Importante salientar que esse valor obtido inicialmente para mse não pode ser convertido diretamente para valor monetário. É necessário aplicar uma segunda função para este fim.

```
41 import numpy as np
42
43 rmse = np.sqrt(metrics.mean_squared_error(steste,previsor))
```

Para isso importamos a biblioteca numpy, na sequência criamos uma variável de nome rmse que aplica a raiz quadrada sobre mse, gerando agora um valor que pode ser levado em conta como unidade monetária.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(200, 5)	Column names: Unnamed: 0, TV, radio, newspaper, sales
entradas	DataFrame	(200, 3)	Column names: TV, radio, newspaper
eteste	DataFrame	(60, 3)	Column names: TV, radio, newspaper
estreino	DataFrame	(140, 3)	Column names: TV, radio, newspaper
mae	float64	1	1.4140186803996229
mse	float64	1	3.2825652471496274
previsor	float64	(60,)	[21.99131061 10.37845701 15.52067254 ... 11.64848 12.8 ...]
rmse	float64	1	1.811785099604704
saidas	Series	(200,)	Series object of pandas.core.series module

Note que agora podemos fazer as devidas comparações. Inicialmente havíamos descoberto que para cada 1 dólar investido, o retorno teria sido de \$1.41, agora, dando mais peso aos erros de processamento, chegamos ao dado de que para cada 1 dólar investido, o retorno foi de \$1.81, o que condiz perfeitamente com os dados plotados lá no início em nossos gráficos.

Código Completo:

```

1 import pandas as pd
2 import seaborn as sns
3 from sklearn.linear_model import LinearRegression
4 from sklearn import metrics
5 import numpy as np
6
7 base = pd.read_csv('Advertising.csv')
8 entradas = base[['TV', 'radio', 'newspaper']]
9 saidas = base['sales']
10
11 sns = sns.pairplot(base,
12                     x_vars = ['TV', 'radio', 'newspaper'],
13                     y_vars = 'sales',
14                     size = 5,
15                     kind = 'reg')
16
17 from sklearn.model_selection import train_test_split
18
19 etreino, eteste, streino, steste = train_test_split(entradas,
20                                                     saidas,
21                                                     test_size = 0.3)
22 reglinear = LinearRegression()
23 reglinear.fit(entreino,streino)
24
25 print(list(zip(['TV', 'radio', 'newspaper'], reglinear.coef_)))
26 print(reglinear.predict([[230.1,37.8,69.2]]))
27
28 previsor = reglinear.predict(eteste)
29 print(previsor)
30
31 mae = metrics.mean_absolute_error(steste,previsor)
32 mse = metrics.mean_squared_error(steste,previsor)
33 rmse = np.sqrt(metrics.mean_squared_error(steste,previsor))

```


32 – Redes Neurais Artificiais Convolucionais

Uma das aplicações mais comuns dentro de machine learning é o processamento de imagens. Na verdade, esta é uma área da computação que tem evoluído a passos largos justamente pelos avanços e inovações que redes neurais artificiais trouxeram para esse meio. O processamento de imagens que se deu início agrupando pixels em posições específicas em tela hoje evoluiu para modelos de visão computacional e até mesmo geração de imagens inteiramente novas a partir de dados brutos de características de imagens usadas para aprendizado.

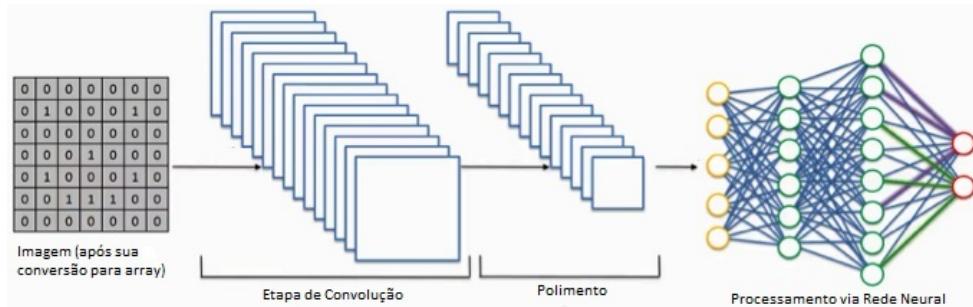
O fato de haver um capítulo inteiro dedicado a este tipo de aprendizado de máquina se dá porque da mesma forma que anteriormente classificamos dados e extraímos informações a partir dos mesmos via redes neurais artificiais, aqui usaremos de modelos de redes neurais que serão adaptados desde o reconhecimento de dígitos e caracteres escritos à mão até modelos capazes de aprender características de um animal e gerar imagens completamente novas a partir de tais características “do que é um cachorro”.

Assim como nos capítulos anteriores, estaremos aprendendo de forma procedural, nos primeiros exemplos contando com bastante referencial teórico para cada linha de código e à medida que você for criando sua bagagem de conhecimento estaremos mais focados a prática.

Como no capítulo anterior, iniciaremos nosso entendimento com base em um modelo que não necessariamente usa de uma rede neural artificial densa para seu processamento, inicialmente temos que fazer uma abordagem que elucide o processo de identificação de uma imagem assim como sua conversão para dados interpretados por um perceptron.

Raciocine que nos próximos exemplos sim estaremos trabalhando com modelos de redes neurais convolucionais, redes muito utilizadas na chamada visão computacional, onde se criam modelos capazes de detectar e reconhecer objetos. Uma das características importantes das redes ditas convolucionais é que apesar de poder haver ou não uma fase de treinamento manual da rede, elas são capazes de aprender sozinhas, reforçando padrões encontrados e a partir desses realizar as conversões de imagem-matriz e matriz-imagem com base em mapeamento de pixels e seus respectivos valores e padrões.

Tenha em mente que aqui abordaremos de forma simples e prática uma das áreas de maior complexidade dentro da computação. Como de costume, abordaremos tudo o que for necessário para entendimento de nossos modelos, porém existe muito a se explorar com base na documentação das ferramentas que usaremos. De forma simples adaptaremos os conceitos aprendidos até então de uma rede neural para operadores de convolução, suas mecânicas de alimentação, conversão e processamento de dados assim como a prática da execução da rede neural convolucional em si.



O processo de processamento de dados oriundos de imagens se dá por uma série de etapas onde ocorrem diversas conversões entre os dados até o formato que finalmente possa ser processado via rede neural. Tenha em mente que a imagem que aparece para o usuário, internamente é uma espécie de mapa de pixels onde é guardado dados de informação como seu aspecto, coloração, intensidade de coloração, posição na grade de resolução, etc... Sendo assim, ainda existe uma segunda etapa de processamento para que tais informações sejam transformadas em padrões e formatos interpretados dentro de uma rede neural para que seja feito o cruzamento desses dados e aplicação das funções necessárias. De forma resumida, por hora, raciocine que uma imagem é lida a partir de algum dispositivo, de imediato serão feitas as conversões pixel a pixel para um mapa, como em um processo de indexação interna, para que tais dados possam ser convertidos para tipo e formato que possa ser processado via rede neural.

Reconhecimento de Caracteres – Digits Dataset

Para darmos início a esta parte dos estudos de redes neurais, iremos fazer o uso de uma base de dados numéricos com 1797 amostras divididas dentro do intervalo de 0 a 9. A ideia é que vamos treinar nosso modelo a aprender as características de cada número e posteriormente conseguir fazer o reconhecimento de um número escrito a mão para teste. Este é um exemplo bastante básico para começarmos a dar início do entendimento de como um computador interpreta imagens para seu processamento.

Nesse exemplo em particular vamos testar dois modelos de processamento para identificação de imagens, pois simularemos aqui um erro bastante comum nesse meio que se dá quando temos que fazer identificação a partir de uma base de dados muito pequena, muito imprecisa que é a execução do modelo confiando em uma margem de precisão relativamente baixa.

```
1 from sklearn import datasets  
2
```

Como sempre, damos início ao processo criando um novo arquivo, nesse caso de nome Digits Dataset.py, em seguida realizamos as primeiras importações necessárias, por hora, importaremos o módulo datasets da biblioteca sklearn. Assim você pode deduzir que inicialmente estaremos usando uma biblioteca de exemplo para darmos os primeiros passos nessa área.

```
3 base = datasets.load_digits()  
4 entradas = base.data  
5 saídas = base.target
```

Logo após criamos nossa variável de nome base, que por sua vez recebe todos os dados do digits dataset por meio da função .load_digits() do módulo datasets. Em seguida criamos nossa variável entradas que recebe os atributos previsores por meio da instância base.data, da mesma forma criamos nossa variável saídas que recebe os dados alvo instanciando base.targets.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
saídas	int32	(1797,)	[0 1 2 ... 8 9 8]

Se o processo de importação ocorreu como esperado foram criadas as devidas variáveis, note que aqui temos a mesma quantidade de dados para nossas entradas e saídas, uma vez que estaremos fazendo o processo de aprendizagem de máquina para que nosso interpretador aprenda a identificar e classificar números escritos a mão. Importante salientar que este número de 1797 amostras é muito pouco quando o assunto é classificação de imagens, o que acarretará em problemas de imprecisão de nosso modelo quando formos executar alguns testes. O valor 64, da segunda coluna de nossa variável entradas diz respeito ao formato em que tais dados estão dispostos, aqui nesse dataset temos 1797 imagens que estão em formato 8x8 pixels, resultando em 64 dados em linhas e colunas que são referências a uma escala de preto e branco.

```
7 print(base.data[0])  
8
```

Para ficar mais claro, por meio da função print() passando como parâmetro a amostra de número 0 de nossa base de dados, poderemos ver via console tal formato.

```
In [ ]: print(base.data[0])
[ 0.  0.  5. 13.  9.  1.  0.  0.  0. 13. 15. 10. 15.  5.  0.  0.  3.
 15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.  8.  0.
 0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 10. 12.
 0.  0.  0.  6. 13. 10.  0.  0.  0.]
```

Via console podemos finalmente visualizar tal formato. Cada valor é referente a uma posição dessa matriz 8x8, onde os valores estão dispostos em um intervalo de 0 até 16, onde 0 seria branco e 16 preto, com valores intermediários nessa escala para cada pixel que compõe essa imagem de um número escrito a mão.

```
9 print(base.images[0])
10
```

Da mesma forma podemos visualizar o formato de imagem salvo na própria base de dados. O método é muito parecido com o anterior, porém instanciamos a amostra número 0 de images.

```
In [ ]: print(base.images[0])
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Via console podemos ver uma matriz 8x8 onde inclusive com o olhar um pouco mais atento podemos identificar que número é esse, referente a primeira amostra da base de dados.

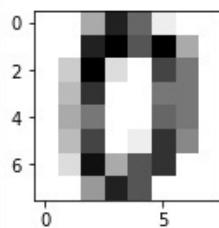
```
11 import matplotlib.pyplot as plt
12
```

Partindo para o que interessa, vamos fazer a plotagem deste número. Para isto, como de praxe, importamos a matplotlib.pyplot.

```
13 plt.figure(figsize = (2,2))
14 plt.imshow(base.images[0],
15            cmap = plt.cm.gray_r)
```

Em seguida, usando apenas o necessário para exibir nossa amostra, criamos essa pequena estrutura onde a função .figure() fará a leitura e plotagem com um tamanho definido em 2x2 (lembreendo que isso não é pixels, apenas uma referência métrica interna a matplotlib). Na sequência a função .imshow() fica responsável por exibir nossa amostra parametrizada também em uma escala de cinza.

```
Out[5]: <matplotlib.image.AxesImage at 0x26db21af668>
```



Não havendo nenhum erro de sintaxe é finalmente exibido em nosso console o número 0, referente a primeira amostra de nossa base de dados. Note a baixa qualidade da imagem, tanto a baixa qualidade quanto o número relativamente pequeno de amostras são um grande problema quando se diz respeito em classificação de imagens.

Revisando, toda imagem, internamente, para nosso sistema operacional e seus interpretadores, é um conjunto de referências a pixels e suas composições, aqui inicialmente estamos trabalhando com imagens de pequeno tamanho e em escala de cinza, mas independente disso, toda imagem possui um formato, a partir deste, uma matriz com dados de tonalidades de cinza (ou de vermelho, verde e azul no caso de imagens coloridas), que em algum momento será convertido para binário e código de máquina. A representação da imagem é uma abstração que o usuário vê em tela, desta disposição de pixels compondo uma figura qualquer.

```
17 from sklearn.model_selection import train_test_split
18 etreino,eteste,streino,steste = train_test_split(entradas,
19                                                 saídas,
20                                                 test_size = 0.1,
21                                                 random_state = 2)
```

Dando sequência, mesmo trabalhando sobre imagens podemos realizar aqueles testes de performance que já são velhos conhecidos nossos. Inicialmente importamos `train_test_split`, criamos suas respectivas variáveis, alimentamos a ferramenta com dados de entrada e de saída, assim como separamos as amostras em 90% para treino e 10% para teste, por fim, definimos que o método de separação das amostras seja randômico não pegando dados com proximidade igual ou menos a 2 números.

```
23 from sklearn import svm
24 classificador = svm.SVC()
25 classificador.fit(entreino,streino)
26 previsor = classificador.predict(eteste)
```

Separadas as amostras, hora de criar nosso modelo inicial de classificador, aqui neste exemplo inicialmente usaremos uma ferramenta muito usada quando o assunto é identificação e classificação de imagens chamada Suport Vector Machine, esta ferramenta está integrada na biblioteca `sklearn` e pode ser facilmente importada como importamos outras ferramentas em outros momentos. Em seguida criamos nosso classificador que inicializa a ferramenta `svm.SVC()`, sem parâmetros mesmo, e codificada nessa sintaxe. Logo após por meio da função `.fit()` alimentamos nosso classificador com os dados de `entreino` e `streino`. Por fim, criamos nosso primeiro previsor, que com base nos dados encontrados em `classificador` fará as devidas comparações e previsões com os dados de `eteste`.

```
28 from sklearn import metrics
29 margem_acerto = metrics.accuracy_score(steste, previsor)
```

Criado o modelo, na sequência como de costume importamos `metrics` e criamos nossa variável responsável por fazer o cruzamento dos dados e o teste de performance do modelo. Então é criada a variável `margem_acerto` que por sua vez executa a ferramenta `.accuracy_score()` tendo como parâmetros os dados de `steste` e os encontrados em `previsor`.

Nome	Tipo	Tamanho	Valor
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
estreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
saidas	int32	(1797,)	[0 1 2 ... 8 9 8]
steste	int32	(180,)	[4 0 9 ... 0 0 4]
streino	int32	(1617,)	[0 6 5 ... 1 1 5]

Selecionado e executado todo bloco de código anterior, via explorador de variáveis podemos nos ater aos dados apresentados nesses testes iniciais. O mais importante deles por hora, margem_acerto encontrou em seu processamento uma taxa de precisão de apenas 56%. Lembre-se que comentei em parágrafos anteriores, isto se dá por dois motivos, base de dados pequena e imagens de baixa qualidade. De imediato o que isto significa? Nossos resultados não são nenhum pouco confiáveis nesse modelo, até podemos realizar as previsões, porém será interessante repetir os testes algumas vezes como prova real, como viés de confirmação das respostas encontradas dentro dessa margem de acertos relativamente baixa.

Leitura e reconhecimento de uma imagem.

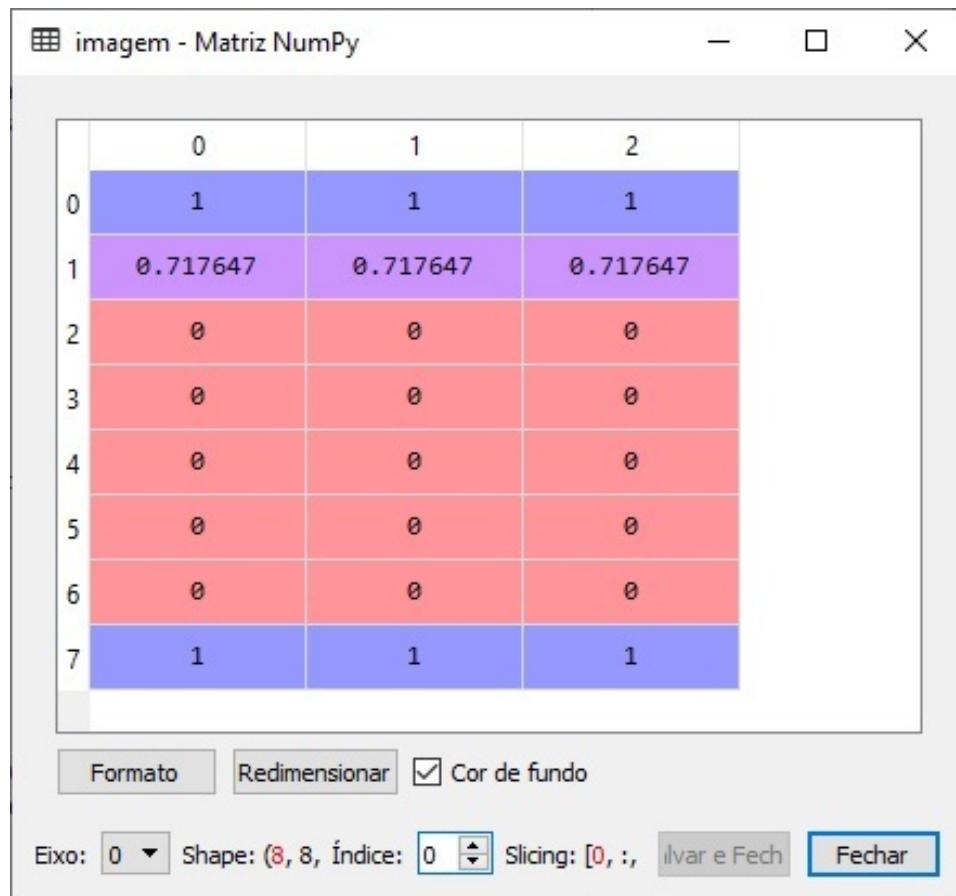
```
31 import numpy as np
32 import matplotlib.image as mimg
```

Uma vez criado o modelo, mesmo identificada sua baixa precisão, podemos realizar testes reais para o mesmo, fazendo a leitura de um número escrito a mão a partir de uma imagem importada para nosso código. Como sabemos o número em questão esse processo pode ser considerado válido como teste, até mesmo com a nossa atual margem de acerto.

Anteriormente havíamos importado a ferramenta pyplot da matplotlib, dessa vez a ferramenta que usaremos é a image, da própria matplotlib, o processo de importação é exatamente igual ao de costume, dessa vez referenciamos a ferramenta como mimg. Por fim, note que também importamos a biblioteca numpy.

```
34 imagem = mimg.imread('num2.png')
35 print(imagem)
```

Dando sequência, criamos nossa variável de nome imagem que recebe como atributo o conteúdo importado de num2.png por meio da função .imread(). A seguir por meio do comando print() imprimimos imagem em nosso console.



Via explorador de variáveis podemos abrir a variável imagem e seu conteúdo, muita atenção ao rodapé da janela, uma vez que estamos visualizando aqui uma matriz, estes dados dispostos nessa grade 8x8 é apenas a informação de 1 pixel da imagem, por meio dos botões da parte inferior da janela podemos navegar pixel a pixel.

```
...: print(imagem)
[[[1.        1.        1.        ]
 [0.7176471 0.7176471 0.7176471]
 [0.        0.        0.        ]
 [0.        0.        0.        ]
 [0.        0.        0.        ]
 [0.        0.        0.        ]
 [0.        0.        0.        ]
 [1.        1.        1.        ]]

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

Via console podemos ver em forma de lista todos dados também referentes a cada pixel.

Raciocine que por hora estes formatos de dados nos impedem de aplicar qualquer tipo de função sobre os mesmos, sendo assim, precisamos fazer a devida conversão desses dados referentes a cada pixel para uma matriz que por sua vez contenha dados em forma de uma matriz, mais especificamente uma array do tipo numpy.

```

37 def rgb2gray(rgb):
38     img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
39     img_array = (16 - (img_array * 16)).astype(int)
40     img_array = img_array.flatten()
41     return img_array

```

Poderíamos criar uma função do zero que fizesse tal processo, porém vale lembrar que estamos trabalhando com Python, uma linguagem com uma enorme comunidade de desenvolvedores, em função disso com apenas uma rápida pesquisa no Google podemos encontrar uma função já pronta que atende nossos requisitos. Analisando a função em si, note que inicialmente é criada a função `rgb2gray` que recebe como parâmetro a variável temporária `rgb`, uma vez que essa será substituída pelos arquivos que faremos a leitura e a devida identificação. Internamente temos uma variável de nome `img_array`, inicialmente ela recebe o produto escalar de todas as linhas e das 3 colunas do formato anterior (matriz de pixel) sobre os valores 0.299, 0.587 e 0.114, uma convenção quando o assunto é conversão para `rgb` (escala colorida red grey blue). Em seguida é feita a conversão desse valor resultado para tipo inteiro por meio da função `.astype()` parametrizada com `int`. Logo após é aplicada a função `.flatten()` que por sua vez irá converter a matriz atual por uma nova indexada no formato que desejamos, 64 valores em um intervalo entre 0 e 16 (escala de cinza) para que possamos aplicar o devido processamento que faremos a seguir.

```

43 rgb2gray(imagem)
44

```

Para ficar mais claro o que essa função faz, chamando-a e passando como parâmetro nossa variável `imagem` (que contém o arquivo lido anteriormente `num2.png`) podemos ver o resultado de todas conversões em nosso console.

```

In [ ]: rgb2gray(imagem)
Out[ ]:
array([ 0,  4, 16, 16, 16, 16, 16,  0,  0, 16,  0,  0,  0,  0,  0,
       0, 16,  0,  0,  0,  0,  0,  0,  7, 16, 16, 16,  6,  0,  0,  0,
       0,  0,  0, 16,  0,  0, 16,  0,  0,  0, 16, 16,  0,  0,  0, 16,
      16, 16,  0,  0,  0,  0, 16, 16,  0,  0])

```

Aplicadas as conversões sobre `num2.png` que estava instanciada em nossa variável `imagem`, podemos notar que temos uma array com 64 referências de tonalidades de cinza em uma posição organizada para o devido reconhecimento.

```

45 identificador = svm.SVC()
46 identificador.fit(entradas,saidas)
47 previsor_id = identificador.predict([rgb2gray(imagem)])
48 print(previsor_id)

```

Criada toda estrutura inicial de nosso modelo, responsável por fazer até então a leitura e a conversão do arquivo de imagem para que seja feito o reconhecimento de caractere, hora de finalmente realizar os primeiros testes de reconhecimento e identificação. Aqui, inicialmente usaremos uma ferramenta chamada Suport Vector Machine. Esta ferramenta por sua vez usa uma série de métricas de processamento vetorial sobre os dados nela alimentados. Uma SVM possui a característica de por meio de aprendizado supervisionado (com base em treinamento via uma base de dados), aprender a reconhecer padrões e a partir destes realizar classificações.

Inicialmente criamos nossa variável de nome `identificador`, que por sua vez inicia o módulo `SVC` da ferramenta `svm`. Em seguida por meio da função `.fit()` é feita a alimentação da mesma com os dados de entrada e saída, lembrando que nesse modelo estamos tratando de atributos previsores e das devidas saídas, já que estamos treinando nossa máquina para reconhecimento de imagens.

Logo após criamos nossa variável de nome previsor_id, que por sua vez via função .predict() recebe nossa função rgb2gray alimentada com nosso arquivo num2.png. Por fim, simplesmente imprimimos em nosso console o resultado obtido em previsor_id.

```
In [ ]: identificador = svm.SVC()
...: identificador.fit(entradas,saidas)
...: previsor_id = identificador.predict([rgb2gray(imagem)])
...: print(previsor_id)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\svm\base.py:193:
FutureWarning: The default value of gamma will change from 'auto' to 'scale' in
version 0.22 to account better for unscaled features. Set gamma explicitly to
'auto' or 'scale' to avoid this warning.
"avoid this warning.", FutureWarning)
[5]
```

Selecionando e executando o bloco de código anterior podemos acompanhar via console o seu processamento e que, apesar de nossa margem de acerto relativamente baixa no processo de aprendizado, ele identificou corretamente que o número passado para identificação (nossa arquivo num2.png) era o número 5 escrito a mão. Da mesma forma podemos realizar mais testes de leitura, desde que alimentando nosso código com imagens no mesmo padrão. Porém, para não nos estendermos muito nesse capítulo inicial, vamos realizar uma segunda verificação independente da SVM, para termos um segundo parâmetro de confirmação. Já sabemos que a leitura e identificação de nosso arquivo foi feita corretamente, mas caso houvesse ocorrido um erro de identificação esse segundo teste poderia nos ajudar de alguma forma. Em outro momento já usamos do modelo de regressão logística para classificar dados quanto seu agrupamento e cruzamento de dados via matriz, aqui nossas conversões vão de um arquivo de imagem para uma array numpy, sendo possível aplicar o mesmo tipo de processamento.

```
50 from sklearn.linear_model import LogisticRegression
51 logr = LogisticRegression()
52 logr.fit(entreino,streino)
53 previsor_logr = logr.predict(eteste)
54 acerto_logr = metrics.accuracy_score(steste, previsor_logr)
55 print(acerto_logr)
```

Como não havíamos usado nada deste modelo em nosso código atual, realizamos a importação da ferramenta LogisticRegression do módulo linear_model da biblioteca sklearn. Em seguida criamos uma variável logr que inicializa a ferramenta, na sequência alimentamos a ferramenta com os dados de entreino e streino, por fim criamos uma variável previsor_logr que faz as previsões sobre os dados de eteste. Também criamos uma variável de nome acerto_logr que mostra a taxa de precisão cruzando os dados de steste e previsor_logr, por fim, exibindo no console este resultado.

Nome	Tipo	Tamanho	Valor
acerto_logr	float64	1	0.9277777777777778
base	utils.Bunch	5	Bunch object of sklearn.utils module
entradas	float64	(1797, 64)	[[0. 0. 5. ... 0. 0. 0.] [0. 0. 0. ... 10. 0. 0.]
eteste	float64	(180, 64)	[[0. 0. 0. ... 0. 0. 0.] [0. 0. 1. ... 15. 1. 0.]
etreino	float64	(1617, 64)	[[0. 0. 10. ... 0. 0. 0.] [0. 0. 0. ... 16. 9. 0.]
imagem	float32	(8, 8, 3)	[[[1. 1. 1.] [0.7176471 0.7176471 0.7176471] ...]
margem_acerto	float64	1	0.5611111111111111
previsor	int32	(180,)	[4 5 9 ... 5 0 5]
previsor_id	int32	(1,)	[5]

Via explorador de variáveis podemos ver que o modelo de regressão logística, aplicado em nossa base de dados, obteve uma margem de acertos de 92%, muito maior aos 56% da SVM que em teoria deveria retornar melhores resultados por ser especializada em reconhecimento e identificação de padrões para imagens. Novamente, vale lembrar que isso se dá porque temos uma base de dados muito pequena e com imagens de baixa qualidade, usadas apenas para fins de exemplo, em aplicações reais, tais margens de acerto tendem a ser o oposto do aqui apresentado.

```
57 regressor = LogisticRegression()
58 regressor.fit(entradas,saidas)
59 previsor_regl = regressor.predict([rgb2gray(imagem)])
60 print(previsor_regl)
```

Para finalizar, criamos nosso regressor por meio de uma variável homônima que inicializa a ferramenta, na sequência é alimentada com os dados contidos em entradas e saídas. Também criamos a variável previsor_regl que realiza as devidas previsões com base na nossa função rgb2gray alimentada com nosso arquivo instanciado num2.png. Concluindo o processo imprimimos em nosso console o resultado obtido para previsor_regl.

```
In [ ]: regressor = LogisticRegression()
...: regressor.fit(entradas,saidas)
...: previsor_regl = regressor.predict([rgb2gray(imagem)])
...: print(previsor_regl)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
432: FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
FutureWarning)
C:\Users\Fernando\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:
469: FutureWarning: Default multi_class will be changed to 'auto' in 0.22. Spec:
the multi_class option to silence this warning.
"this warning.", FutureWarning)
[5]
```

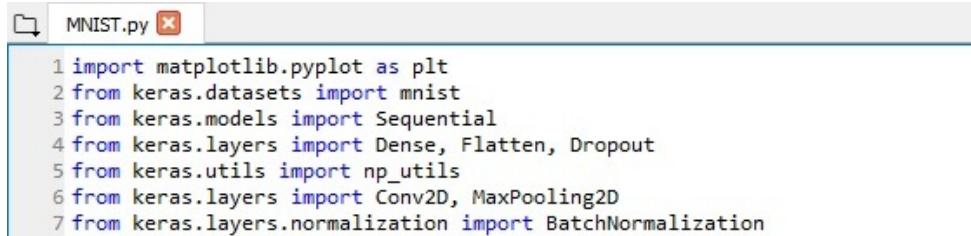
Via console podemos acompanhar o processamento e como esperado, a previsão da identificação do arquivo, reconhecido como número 5 escrito a mão.

Código Completo:

```
1 from sklearn import datasets
2 import matplotlib.pyplot as plt
3 from sklearn.model_selection import train_test_split
4 from sklearn import svm
5 from sklearn import metrics
6 import numpy as np
7 import matplotlib.image as mimg
8 from sklearn.linear_model import LogisticRegression
9
10 base = datasets.load_digits()
11 entradas = base.data
12 saidas = base.target
13
14 plt.figure(figsize = (2,2))
15 plt.imshow(base.images[0],
16             cmap = plt.cm.gray_r)
17
18 etreino,eteste,streino,steste = train_test_split(entradas,
19                                                 saidas,
20                                                 test_size = 0.1,
21                                                 random_state = 2)
22 classificador = svm.SVC()
23 classificador.fit(etreino,streino)
24 previsor = classificador.predict(eteste)
25 margem_acerto = metrics.accuracy_score(steste, previsor)
26
27 imagem = mimg.imread('num2.png')
28
29 def rgb2gray(rgb):
30     img_array = np.dot(rgb[...,:3],[0.299,0.587,0.114])
31     img_array = (16 - (img_array * 16)).astype(int)
32     img_array = img_array.flatten()
33     return img_array
34
35 rgb2gray(imagem)
36
37 identificador = svm.SVC()
38 identificador.fit(entradas,saidas)
39 previsor_id = identificador.predict([rgb2gray(imagem)])
40 print(previsor_id)
41
42 logr = LogisticRegression()
43 logr.fit(etreino,streino)
44 previsor_logr = logr.predict(eteste)
45 acerto_logr = metrics.accuracy_score(steste, previsor_logr)
46 print(acerto_logr)
47
48 regressor = LogisticRegression()
49 regressor.fit(entradas,saidas)
50 previsor_regl = regressor.predict([rgb2gray(imagem)])
51 print(previsor_regl)
```


Classificação de Dígitos Manuscritos – MNIST Dataset

Dados os nossos passos iniciais nesta parte de identificação e reconhecimento de dígitos manuscritos, hora de fazer o mesmo tipo de processamento agora de forma mais robusta, via rede neural artificial. Desta vez estaremos usando o dataset MNIST, base de dados integrante da biblioteca Keras, com maiores amostras de imagem e imagens também armazenadas em uma resolução maior. Lembre-se que no exemplo anterior nosso grande problema foi trabalhar sobre uma baixa margem de precisão devido os dados de nossa base que não colaboravam para um melhor processamento.



```
1 import matplotlib.pyplot as plt
2 from keras.datasets import mnist
3 from keras.models import Sequential
4 from keras.layers import Dense, Flatten, Dropout
5 from keras.utils import np_utils
6 from keras.layers import Conv2D, MaxPooling2D
7 from keras.layers.normalization import BatchNormalization
```

Sem mais delongas vamos direto ao ponto, direto ao código. Como sempre, inicialmente criamos um novo arquivo de nome MNIST.py que armazenará nossos códigos, dentro de si começamos realizando as devidas importações.

Não muito diferente do que estamos habituados, realizamos a importação da matplotlib.pyplot (para plotagem/exibição das imagens via console), da base de dados mnist, das ferramentas Sequential, Dense, Flatten e Dropout de seus referidos módulos. Também importamos np_utils (para nossas transformações de formato de matriz), em seguida importamos Conv2D (ferramenta pré-configurada para criação de camadas de rede neural adaptada a processamento de dados de imagem), MaxPooling2d (ferramenta que criará micromatrizes indexadas para identificação de padrões pixel a pixel) e por fim BatchNormalization (ferramenta que realiza uma série de processos internos para diminuir a margem de erro quando feita a leitura pixel a pixel da imagem).

```
9 (eteste,streino),(eteste,steste) = mnist.load_data()
10
```

Na sequência criamos as variáveis que armazenarão os dados de entrada e de saída importados da base mnist.

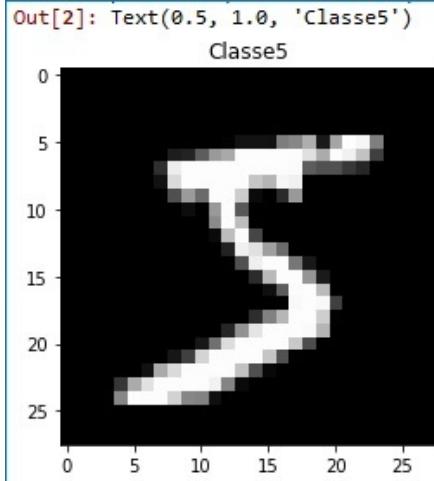
Nome	Tipo	Tamanho	Valor
eteste	float32	(10000, 28, 28, 1)	[[[[0.] [0.]
estreino	float32	(60000, 28, 28, 1)	[[[[0.] [0.]
steste	float32	(10000, 10)	[[0. 0. 0. ... 1. 0. 0.] [0. 0. 1. ... 0. 0. 0.]
streino	float32	(60000, 10)	[[0. 0. 0. ... 0. 0. 0.] [1. 0. 0. ... 0. 0. 0.]

Fazendo uma rápida leitura via explorador de variáveis podemos ver que de fato nossas variáveis foram criadas. Note que imediatamente podemos deduzir o tamanho dessa base de dados, uma vez que temos 60000 amostras separadas para treino e 10000 separadas para teste. Um número consideravelmente maior do que a base usada no exemplo anterior (1797 amostras apenas). Repare nos dados de entrada que eles possuem uma configuração adicional (28,28,1), que

significa que cada amostra está numa configuração de matrix 28x28 de uma camada de profundidade (pode existir processamento sobre voxels, onde haverão mais camadas de profundidade/volume).

```
11 plt.imshow(etreino[0], cmap = 'gray')
12 plt.title('Classe' + str(streino[0]))
```

Em seguida fazendo uma simples plotagem, nos mesmos moldes do modelo anterior, pegamos a amostra de índice número 0 de nossa base de entradas para treino, apenas por conformidade instanciamos a nomenclatura da amostra de índice número 0 de nossa base de saída de treino.



Selecionado e executado o bloco de código anterior, podemos ver via console que das amostras de saída categorizadas como “número 5”, a imagem apresentada é um número 5 que podemos deduzir pelo seu formato ter sido escrito à mão. Repare a qualidade da imagem, muito superior em relação a do exemplo anterior.

```
14 etreino = etreino.reshape(etreino.shape[0],28,28,1)
15 eteste = eteste.reshape(eteste.shape[0],28,28,1)
```

Iniciando a fase de polimento de nossos dados, o que faremos em cima de uma base de dados de imagens de números é converter essas informações de mapeamento de pixels para um formato array onde se possa ser aplicado funções. Sendo assim, criamos nossa variável de nome etreino que recebe sobre si a função .reshape(), redimensionando todas amostras para o formato 28x28 de 1 camada. O mesmo é feito para eteste.

Nome	Tipo	Tamanho	Valor
eteste	uint8	(10000, 28, 28, 1)	<code>[[[[]]]</code> [0]
etreino	uint8	(60000, 28, 28, 1)	<code>[[[[]]]</code> [0]
steste	uint8	(10000,)	[7 2 1 ... 4 5 6]
streino	uint8	(60000,)	[5 0 4 ... 5 6 8]

Repare que as variáveis etreino e eteste foram atualizadas para o novo formato.

```
17 etreino = etreino.astype('float32')
18 eteste = eteste.astype('float32')
```

Em seguida realizamos uma nova alteração e atualização sobre etreino e eteste, dessa vez por meio da função .astype() convertemos o seu conteúdo para o tipo float32 (números com casas

decimais).

```
20 etreino /= 255
21 eteste /= 255
```

Prosseguindo alteramos e atualizamos etreino e eteste mais um vez, dessa vez dividindo o valor de cada amostra por 255.

```
23 streino = np_utils.to_categorical(streino, 10)
24 steste = np_utils.to_categorical(steste, 10)
```

Finalizada a formatação de nossos dados de entrada, hora de realizar o polimento sobre os dados de saída. Aqui simplesmente aplicamos a função `.to_categorical()` que altera e atualiza os dados em 10 categorias diferentes (números de 0 a 9).

Nome	Tipo	Tamanho	Valor
eteste	float32	(10000, 28, 28, 1)	[[[0.] [0.] [0.]
etreino	float32	(60000, 28, 28, 1)	[[[0.] [0.] [0.]
steste	float32	(10000, 10)	[[0. 0. 0. ... 1. 0. 0.] [0. 0. 1. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]
streino	float32	(60000, 10)	[[0. 0. 0. ... 0. 0. 0.] [1. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]

Feito o reajuste de nossos dados de saída temos agora dados matriciais compatíveis tanto para o cruzamento entre eles quanto para aplicação de funções. Note que terminamos com 60000 amostras que servirão para treinar nossa rede neural, realizaremos testes sobre 10000 amostras dessas, as categorizando em números de 0 a 9.

```
26 classificador = Sequential()
27 classificador.add(Conv2D(32,
28     (3,3),
29     input_shape = (28,28,1),
30     activation = 'relu'))
31 classificador.add(BatchNormalization())
32 classificador.add(MaxPooling2D(pool_size = (2,2)))
33 classificador.add(Conv2D(32,
34     (3,3),
35     activation = 'relu'))
```

Agora damos início a criação da estrutura de nossa rede neural artificial convolucional, você verá que o modelo em si segue um certo padrão que você já está familiarizado, apenas acrescentando ou alterando algumas funções que serão responsáveis pelo processamento e detecção de padrões sobre esses dados providos de imagens.

Dessa forma iniciamos o modelo criando nosso classificador, que inicialmente só executa a ferramenta Sequential, em seguida criamos a camada de entrada e nossa primeira camada oculta, note que dessa vez não estamos usando Dense para isso, mas Conv2D que recebe como parâmetro `input_shape = (28,28,1)`, enquanto anteriormente definímos um número de neurônios para camada de entrada, agora estamos definindo que a camada de entrada é uma matrix 28x28 de uma camada de profundidade, o método de ativação da mesma é 'relu' e os parâmetros referentes a primeira camada oculta, que nesse caso é o chamado operador de convolução que irá pegar os dados brutos da matriz de entrada, realizar a detecção de suas características, parear com uma nova matriz aplicando funções pixel a pixel e por fim gerando uma série de mapas de características. Esta é uma forma de redimensionar a imagem, diminuindo seu tamanho e convertendo as informações relevantes de padrões de pixels que compõe a imagem para que seu processamento seja facilitado. Em seguida é realizado o processo de normalização desses dados,

você já deve ter notado que em imagens de baixa qualidade, em torno do traço existem pixels borrados, imprecisos, aqui o operador simplesmente verificará qual a relevância desses pixels e, sempre que possível, irá eliminar do mapa os desnecessários para reduzir ainda mais a quantidade de dados a serem processados. Repare que aqui via MaxPooling é feito um novo redimensionamento para uma matrix 2x2, contendo um mapa com apenas as características relevantes a imagem. O que deve ficar bem entendido aqui é que não estamos simplesmente redimensionando uma imagem reduzindo seu tamanho de largura e altura, estamos criando um sistema de indexação interno que pegará a informação de cada pixel e sua relação com seus vizinhos, gerando um mapa de características e reduzindo o número de linhas e colunas de sua matriz, no final do processo temos um mapeamento que consegue reconstruir a imagem original (ou algo muito próximo a ela) mas que para processamento via rede neural possui um formato matricial pequeno e simples. Como sempre, a segunda camada oculta agora não conta mais com o parâmetro dos dados de entrada.

```

36 classificador.add(BatchNormalization())
37 classificador.add(MaxPooling2D(pool_size = (2,2)))
38 classificador.add(Flatten())
39 classificador.add(Dense(units = 128,
40                         activation = 'relu'))
41 classificador.add(Dropout(0.2))
42 classificador.add(Dense(units = 128,
43                         activation = 'relu'))
44 classificador.add(Dropout(0.2))
45 classificador.add(Dense(units = 10,
46                         activation = 'softmax'))
```

Dando sequência após algumas camadas de processamento, redimensionando os dados de entrada, é executada a ferramenta Flatten que por sua vez irá pegar essas matrizes e seus referentes mapas de características e irá transformar isto em uma lista, onde cada valor (de cada linha) atuará como um neurônio nesta camada intermediária, finalizada essa etapa agora criamos camadas densas, aplicamos dropout em 20% (ou seja, de cada camada o operador pegará 20% dos neurônios e simplesmente irá subtraí-los da camada, a fim de também diminuir seu tamanho) e assim chegamos a camada de saída, com 10 neurônios (já que estamos classificando dados em números de 0 a 9) que por sua vez tem ativação ‘softmax’, função de ativação essa muito parecida com a nossa velha conhecida ‘sigmoid’, porém capaz de categorizar amostras em múltiplas saídas.

```

47 classificador.compile(loss = 'categorical_crossentropy',
48                       optimizer = 'adam',
49                       metrics = ['accuracy'])
50 classificador.fit(etreino,
51                   streino,
52                   batch_size = 128,
53                   epochs = 10,
54                   validation_data = (eteste,steste))
```

Por fim, criamos a estrutura que compila a rede, definindo sua função de perda como ‘categorical_crossentropy’ que como já visto anteriormente, verifica quais dados não foram possíveis de classificar quanto sua proximidade aos vizinhos, otimizador ‘adam’ e como métrica foco em precisão por meio do parâmetro ‘accuracy’. Última parte, por meio da função .fit() alimentamos a rede com os dados de etreino e streino, definimos que os valores dos pesos serão atualizados a cada 128 amostras, que executaremos a rede 10 vezes e que haverá um teste de validação comparando os dados encontrados com os de eteste e steste. Selecionado todo bloco de código, se não houver nenhum erro de sintaxe a rede começará a ser executada. Você irá reparar

que, mesmo com todas reduções que fizemos, e mesmo executando nosso modelo apenas 10 vezes, o processamento dessa rede se dará por um bom tempo, dependendo de seu hardware, até mesmo mais de uma hora.

```
60000/60000 [=====] - 257s 4ms/step - loss: 0.0259 - acc: 0.9921 - val_loss: 0.0305 - val_acc: 0.9904
Epoch 7/10
60000/60000 [=====] - 249s 4ms/step - loss: 0.0230 - acc: 0.9928 - val_loss: 0.0364 - val_acc: 0.9894
Epoch 8/10
60000/60000 [=====] - 294s 5ms/step - loss: 0.0214 - acc: 0.9932 - val_loss: 0.0287 - val_acc: 0.9910
Epoch 9/10
60000/60000 [=====] - 278s 5ms/step - loss: 0.0165 - acc: 0.9948 - val_loss: 0.0358 - val_acc: 0.9912
Epoch 10/10
60000/60000 [=====] - 255s 4ms/step - loss: 0.0162 - acc: 0.9953 - val_loss: 0.0364 - val_acc: 0.9902
```

Terminada a execução da rede note que, feito todo processamento de todas camadas e parâmetros que definimos chegamos a excelentes resultados, onde a função de perda nos mostra que apenas 1% dos dados não puderam ser classificados corretamente, enquanto val_acc nos retorna uma margem de acertos/precisão de 99% neste tipo de classificação.

Código Completo:

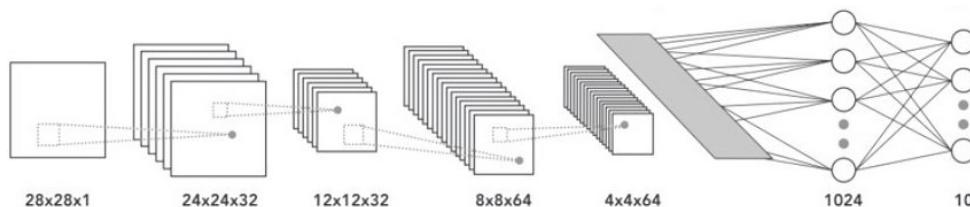
```
1 import matplotlib.pyplot as plt
2 from keras.datasets import mnist
3 from keras.models import Sequential
4 from keras.layers import Dense, Flatten, Dropout
5 from keras.utils import np_utils
6 from keras.layers import Conv2D, MaxPooling2D
7 from keras.layers.normalization import BatchNormalization
8
9 (etreino,streino),(eteste,steste) = mnist.load_data()
10
11 plt.imshow(etreino[0], cmap = 'gray')
12 plt.title('Classe' + str(streino[0]))
13
14 etreino = etreino.reshape(etreino.shape[0],28,28,1)
15 eteste = eteste.reshape(eteste.shape[0],28,28,1)
16
17 etreino = etreino.astype('float32')
18 eteste = eteste.astype('float32')
19
20 etreino /= 255
21 eteste /= 255
22
23 streino = np_utils.to_categorical(streino, 10)
24 steste = np_utils.to_categorical(steste, 10)
```

```

26 classificador = Sequential()
27 classificador.add(Conv2D(32,
28     (3,3),
29     input_shape = (28,28,1),
30     activation = 'relu'))
31 classificador.add(BatchNormalization())
32 classificador.add(MaxPooling2D(pool_size = (2,2)))
33 classificador.add(Conv2D(32,
34     (3,3),
35     activation = 'relu'))
36 classificador.add(BatchNormalization())
37 classificador.add(MaxPooling2D(pool_size = (2,2)))
38 classificador.add(Flatten())
39 classificador.add(Dense(units = 128,
40     activation = 'relu'))
41 classificador.add(Dropout(0.2))
42 classificador.add(Dense(units = 128,
43     activation = 'relu'))
44 classificador.add(Dropout(0.2))
45 classificador.add(Dense(units = 10,
46     activation = 'softmax'))
47 classificador.compile(loss = 'categorical_crossentropy',
48     optimizer = 'adam',
49     metrics = ['accuracy'])
50 classificador.fit(etreino,
51     streino,
52     batch_size = 128,
53     epochs = 10,
54     validation_data = (eteste,steste))

```

Aqui quero apenas fazer um breve resumo porque acho necessário para melhor entendimento. Sem entrar em detalhes da estrutura de nosso modelo, raciocine que o processo de classificação de imagens é bastante complexo, demanda muito processamento, uma vez que a grosso modo, uma imagem que temos em tela é simplesmente um conjunto de pixels em suas devidas posições com suas respectivas cores. Internamente, para que possamos realizar processamento via rede neural artificial, devemos fazer uma série de transformações/conversões para que essa imagem se transforme em uma série de mapas matriciais onde possam se aplicar funções. Para isso, temos que fazer o redimensionamento de tal imagem, sem perda de informação, de uma matriz grande para pequena, que será convertida em dados de camada para a rede neural, por fim, a rede irá aprender a reconhecer as características e padrões de tais matrizes e reconstruir toda essa informação novamente em imagem.



Infográfico mostrando desde a leitura da imagem em sua forma matricial, geração de mapas e polimento, conversão em neurônios e interação com camada subsequente.

Lembrando que este processo é apenas a estrutura da rede, assim como nos modelos anteriores, uma vez que nossa rede esteja funcionando podemos aplicar testes e fazer previsões em cima do mesmo.

```
1 from keras.datasets import mnist
2 from keras.models import Sequential
3 from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4 from keras.utils import np_utils
5 import numpy as np
6 from sklearn.model_selection import StratifiedKFold
```

Muito bem, realizado o processamento convencional de nossa rede neural convolucional, hora de aplicar algum teste para verificar a integridade de nossos resultados. Neste tipo de rede neural um dos testes mais eficazes é o nosso velho conhecido teste de validação cruzada, onde podemos separar nossa base de dados em partes e testá-las individualmente. Aqui, lembrando que temos um modelo que está processando dados a partir de imagens, usaremos uma ferramenta um pouco diferente para o mesmo fim, a StratifiedKFold da biblioteca sklearn. O processo das importações é o mesmo de sempre.

```
8 seed = 5
9 np.random.seed(seed)
```

Na sequência criamos uma variável de nome seed com valor inicial 5, que em sua função subsequente fará a alimentação de nossa ferramenta StratifiedKFold com valores randômicos dentro desse intervalo.

```
11 (entreino,streino), (eteste,stteste) = mnist.load_data()
12 entradas = entreino.reshape(entreino.shape[0], 28, 28, 1)
13 entradas = entradas.astype('float32')
14 entradas /= 255
15 saidas = np_utils.to_categorical(streino, 10)
```

Logo após realizamos aquele procedimento de importação dos dados de nossa base de dados assim como a distribuição desses dados em variáveis de treino, teste, entrada e de saída. Assim como as devidas conversões de formato da mesma forma que realizamos anteriormente na criação de nosso modelo convolucional.

```
17 kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
18 resultados = []
```

Em seguida criamos uma variável de nome kfold que inicializa a ferramenta StratifiedKFold, note que definimos os parâmetros n_splits = 5, que significa que a base será dividida em 5 partes iguais, shuffle = True define que as amostras sejam pegas de forma aleatória, para evitar testes “viciados” e random_state que recebe o valor 5 que havíamos definido anteriormente em seed.

```
20 a = np.zeros(5)
21 b = np.zeros(shape = (saidas.shape[0], 1))
```

Na etapa seguinte criamos duas variáveis a e b que por meio da função `.zeros()` cria arrays preenchidas com números 0 a serem substituídos durante a aplicacão de alguma função.

```
23 for evalcruzada,svalcruzada in kfold.split(entradas,
24                                         np.zeros(shape=(saidas.shape[0],1))
25     classificador = Sequential()
26     classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='relu'))
27     classificador.add(MaxPooling2D(pool_size = (2,2)))
28     classificador.add(Flatten())
29     classificador.add(Dense(units = 128, activation = 'relu'))
30     classificador.add(Dense(units = 10, activation = 'softmax'))
31     classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
32                           metrics = ['accuracy'])
33     classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
34                       batch_size = 128, epochs = 5)
35     precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzada])
36     resultados.append(precisao[1])
```

Dando sequência criamos a estrutura de nosso teste de validação cruzada dentro de um laço de repetição, onde são criadas as variáveis temporárias evalcruzada e svalcruzada referentes aos dados para entrada e saída. Por meio da função kfold.split() então é feita a divisão e separação de acordo com o formato que havíamos parametrizado anteriormente. Note que dentro existe uma estrutura de rede neural assim como as anteriores, apenas agora condensada para poupar linhas aqui do livro. Esta rede por sua vez é alimentada com os dados que forem passados vindos de entradas e saídas sobre evalcruzada assim como seu teste de performance é realizado com os dados atribuídos para entradas e saídas sobre svalcruzada. Por fim o retorno de todo esse processamento (lembrando que aqui nessa fase a rede será executada novamente) irá alimentar resultados por meio da função .append() (uma vez que resultados está em formato de lista).

```
Epoch 2/5
48000/48000 [=====] - 32s 665us/step - loss: 0.0769 -
acc: 0.9781
Epoch 3/5
48000/48000 [=====] - 36s 754us/step - loss: 0.0511 -
acc: 0.9844
Epoch 4/5
48000/48000 [=====] - 33s 685us/step - loss: 0.0366 -
acc: 0.9890
Epoch 5/5
48000/48000 [=====] - 33s 677us/step - loss: 0.0264 -
acc: 0.9923
12000/12000 [=====] - 3s 244us/step
In [ ]:
```

Via console podemos ver que após as novas execuções da rede, agora cruzando dados de amostras aleatórias, houve uma margem de acerto de 99%, compatível com o resultado obtido na rede neural do modelo.

```
38 media = sum(resultados) / len(resultados)
```

Por fim é criada a variável media que realiza o cruzamento e divisão dos dados obtidos nas linhas e colunas de resultados.

Nome	Tipo	Tamanho	Valor
a	float64	(5,)	[0. 0. 0. 0. 0.]
b	float64	(60000, 1)	[[0.] [0.]
entradas	float32	(60000, 28, 28, 1)	[[[[0.]] [0.]]
eteste	uint8	(10000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]
etreino	uint8	(60000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]
evalcruzada	int32	(48000,)	[0 1 3 ... 59997 59998 59999]
media	float64	1	0.98415
precisao	list	2	[0.05134425956034101, 0.98475]

Via explorador de variáveis é possível visualizar um resultado quase idêntico ao da execução da validação cruzada, isso é perfeitamente normal desde que os valores sejam muito aproximados (isto se dá porque em certas métricas de visualização de dados pode ocorrer “arredondamentos” que criam uma variação para mais ou para menos, mas sem impacto nos resultados), 98% de margem de acerto no processo de classificação.

Código Completo (Validação Cruzada):

```
1 from keras.datasets import mnist
2 from keras.models import Sequential
3 from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
4 from keras.utils import np_utils
5 import numpy as np
6 from sklearn.model_selection import StratifiedKFold
7
8 seed = 5
9 np.random.seed(seed)
10
11 (entreino,streino), (eteste,steste) = mnist.load_data()
12 entradas = entreino.reshape(entreino.shape[0], 28, 28, 1)
13 entradas = entradas.astype('float32')
14 entradas /= 255
15 saidas = np_utils.to_categorical(streino, 10)
16
17 kfold = StratifiedKFold(n_splits = 5, shuffle = True, random_state = seed)
18 resultados = []
19
20 a = np.zeros(5)
21 b = np.zeros(shape = (saidas.shape[0], 1))
22
23 for evalcruzada,svalcruzada in kfold.split(entradas,
24                                         np.zeros(shape=(saidas.shape[0],1)))
25
26     classificador = Sequential()
27     classificador.add(Conv2D(32, (3,3), input_shape=(28,28,1), activation='relu'))
28     classificador.add(MaxPooling2D(pool_size = (2,2)))
29     classificador.add(Flatten())
30     classificador.add(Dense(units = 128, activation = 'relu'))
31     classificador.add(Dense(units = 10, activation = 'softmax'))
32     classificador.compile(loss = 'categorical_crossentropy', optimizer='adam',
33                           metrics = ['accuracy'])
34     classificador.fit(entradas[evalcruzada], saidas[evalcruzada],
35                       batch_size = 128, epochs = 5)
36     precisao = classificador.evaluate(entradas[svalcruzada], saidas[svalcruzada])
37     resultados.append(precisao[1])
38 media = sum(resultados) / len(resultados)
```


Classificação de Imagens de Animais – Bichos Dataset

Avançando um pouco mais com nossos estudos sobre redes neurais convolucionais, foi entendido o processo de transformação de uma imagem para formatos de arquivos suportados pelas ferramentas de redes neurais. Também fomos um breve entendimento sobre a forma como podemos treinar um modelo para que este aprenda a identificar características e a partir desse aprendizado, realizar previsões, porém até o momento estávamos nos atendo a modelos de exemplo das bibliotecas em questão.

Na prática, quando você precisar criar um identificador e classificador de imagem este será de uma base de objetos muito diferente dos exemplos anteriores. Em função disso neste tópico vamos abordar o processo de aprendizado de máquina com base em um conjunto de imagens de gatos e cachorros, posteriormente nossa rede será treinada para a partir de seu modelo, identificar novas imagens que fornecermos classificando-as como gato ou cachorro. Note que agora para tentarmos trazer um modelo próximo a realidade, dificultaremos de propósito esse processo, uma vez que estamos classificando 2 tipos de animais esteticamente bastante parecidos.

Nome	Data de modificaç...	Tipo	Tamanho
dataset	19/08/2019 16:05	Pasta de arquivos	
Bichos Dataset.py	19/08/2019 15:24	Arquivo PY	3 KB

O primeiro grande diferencial deste modelo que estudaremos agora é que o mesmo parte do princípio de classificar gatos e cachorros a partir de imagens separadas de gatos e cachorros obviamente, mas completamente aleatórias, de diferentes tamanhos com diferentes características, e não padronizadas como nos exemplos anteriores, dessa forma, essa base de dados que criaremos se aproxima mais das aplicações reais desse tipo de rede neural. No mesmo diretório onde criamos nosso arquivo de código extraímos a base de dados.

Nome	Data de modificaç...	Tipo	Tamanho
test_set	19/08/2019 16:05	Pasta de arquivos	
training_set	19/08/2019 16:05	Pasta de arquivos	
.DS_Store	23/06/2018 13:15	Arquivo DS_STORE	7 KB

Dentro de dataset podemos verificar que temos duas pastas referentes a base de treino e de teste.

Nome	Data de modificaç...	Tipo	Tamanho
cachorro	19/08/2019 16:05	Pasta de arquivos	
gato	19/08/2019 16:05	Pasta de arquivos	
.DS_Store	21/06/2018 19:31	Arquivo DS_STORE	9 KB

Abrindo training_set podemos verificar que existem as categorias cachorro e gato.



Abrindo cachorros note que temos um conjunto de dados de imagens de cachorros diversos (2000 imagens para ser mais exato) com características diversas (tamanho, formato, pelagem, etc...) e em ações diversas (posição na foto). O interessante desse tipo de dataset é que existe muita coisa desnecessária em cada foto, objetos, pessoas, legenda, etc... e nossa rede neural terá de identificar o que é característica apenas do cachorro na foto, para dessa forma realizar previsões posteriormente.

Outro ponto interessante é que esse modelo é facilmente aplicável para classificação de qualquer tipo de objeto, e para classificação de diferentes tipos de objeto (mais de 2), a nível de base de dados a única alteração seria a criação de mais pastas (aqui temos duas, gatos e cachorros) referentes a cada tipo de objeto a ser classificado.

```
File Bichos Dataset.py
1 from keras.models import Sequential
2 from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
3 from keras.layers.normalization import BatchNormalization
4 from keras.preprocessing.image import ImageDataGenerator
```

Como sempre, o processo se dá iniciando com as importações das bibliotecas, módulos e ferramentas que estaremos usando ao longo do código. Dessa forma criamos um novo arquivo de código de nome Bichos Dataset e damos início as importações das bibliotecas já conhecidas, única diferença é que para este modelo importamos do módulo preprocessing da biblioteca keras a ferramenta ImageDataGenerator. Esta ferramenta por sua vez será usada para gerar dados a partir de padrões reconhecidos. Em outras palavras, estaremos usando uma base de dados relativamente pequena, à medida que a rede for encontrando padrões que ela tem certeza ser de gato ou certeza ser de cachorro, ela irá gerar uma nova imagem com indexamento próprio com tais características, facilitando o processo de reforço de aprendizado. A ideia desta ferramenta é compensar a alta complexidade encontrada em processamento de imagens, identificar caracteres como no modelo anterior é muito fácil, identificar características minúsculas, a partir de imagens tão variadas de animais (que por sua vez possuem muitas características variáveis) é um processo bastante complexo até mesmo para redes neurais, sendo assim, nas etapas onde são aplicados métodos de aprendizagem por reforço, a rede usará de características identificadas e separadas

para ganhar eficiência em seu processamento. Na fase de executar a rede propriamente dita você verá que este modelo de rede neural sempre começa com uma margem de acertos bastante baixa e época a época ela aumenta e muito sua precisão, isto se dá por funções internas de ferramentas como a ImageDataGenerator.

```
6 gerador_treino = ImageDataGenerator(rescale = 1.0/255,
7                               rotation_range = 7,
8                               horizontal_flip = True,
9                               shear_range = 0.2,
10                              height_shift_range = 0.07,
11                              zoom_range = 0.2)
12 gerador_teste = ImageDataGenerator(rescale = 1.0/255)
```

Feitas as importações de forma correta podemos dar início diretamente na criação de nosso gerador, em algumas literaturas essa fase é chamada Augmentation, pois estamos justamente junto ao nosso polimento inicial dos dados, realizando o aumento de nossa base para que se tenha melhor precisão no processamento das imagens. Para isso criamos nossa variável gerador_treino que inicializa a ferramenta ImageDataGenerator que por sua vez recebe uma série de parâmetros. O primeiro deles rescale = 1.0/255 nada mais é do que o escalonamento da imagem, rotation_range diz respeito ao fator de rotação (no sentido de angular a imagem nos dois sentidos por um ângulo pré-determinado e salvar também essas informações, horizontal_flip = True, que irá espelhar a imagem e guardar as informações da mesma espelhada, height_shift_range irá encontrar e salvar as informações referentes a possível distorção vertical da imagem e zoom_range por sua vez irá aplicar um zoom de 2% e salvar essa informação. Note que dessa forma, não são apenas convertidos os dados brutos referentes a cada mapeamento de pixel normal, mas essas informações adicionais, quando salvas em blocos, ajudarão o interpretador e eliminar distorções e encontrar padrões mais precisos. Já para nossa variável gerador_teste é feito simplesmente a conversão de escala.

```
14 base_treino = gerador_treino.flow_from_directory('dataset/training_set',
15                                                 target_size = (64,64),
16                                                 batch_size = 32,
17                                                 class_mode = 'binary')
18 base_teste = gerador_teste.flow_from_directory('dataset/test_set',
19                                                 target_size = (64,64),
20                                                 batch_size = 32,
21                                                 class_mode = 'binary')
```

Em seguida são criadas as variáveis referentes as nossas variáveis de entradas e saídas com base nas importações das imagens. Inicialmente criamos a variável base_treino que instancia e executa nosso gerador_treino executando sua função .flow_from_directory() que por sua vez é capaz de ler os arquivos que estão dentro de um diretório específico, lembre-se que nossa base de dados está dividida em imagens de gatos e cachorros divididas para treino e teste dentro de pastas separadas. Como parâmetros, inicialmente é passado o caminho da pasta onde se encontram os arquivos, em nosso caso, ‘dataset/training_set’, na sequência target_size realiza a primeira conversão para uma matriz de mapeamento 64x64, onde batch_size define a criação de blocos de 32 em 32 pixels, convertendo para binário via class_mode. Exatamente o mesmo processo é feito para nossa base_teste, apenas modificando o caminho do diretório onde se encontram as imagens separadas para teste.

```

23 classificador = Sequential()
24 classificador.add(Conv2D(32,
25     (3,3),
26     input_shape = (64,64,3),
27     activation = 'relu'))
28 classificador.add(BatchNormalization())
29 classificador.add(MaxPooling2D(pool_size = (2,2)))
30 classificador.add(Conv2D(32,
31     (3,3),
32     activation = 'relu'))
33 classificador.add(BatchNormalization())
34 classificador.add(MaxPooling2D(pool_size = (2,2)))
35 classificador.add(Flatten())
36 classificador.add(Dense(units = 128,
37     activation = 'relu'))
38 classificador.add(Dropout(0.2))
39 classificador.add(Dense(units = 128,
40     activation = 'relu'))
41 classificador.add(Dropout(0.2))
42 classificador.add(Dense(units = 1,
43     activation = 'sigmoid'))

```

Prosseguindo com nosso código hora de criar a estrutura de rede neural, onde codificamos todo aquele referencial teórico para um formato lógico a ser processado. Inicialmente criamos nosso classificador que inicializa Sequential sem parâmetros. Em seguida é criada a primeira parte de nossa etapa de convolução, onde adicionamos uma camada onde parametrizamos que como neurônios da camada de entrada, temos uma estrutura matricial de 54 por 64 pixels em 3 camadas, que será dividida em blocos de 3x3 resultando em uma primeira camada oculta de 32 neurônios, passando pela ativação relu. Em seguida é adicionada uma camada de normalização por meio da função BatchNormalization, sem parâmetros mesmo. Logo após é adicionada uma camada de polimento onde a matrix anterior 3x3 passa a ser um mapa 2x2. A seguir é repetida esta primeira etapa até o momento da dição da camada Flatten, que por sua vez irá pegar todo resultado da fase de convolução e converter para uma indexação em forma de lista onde cada valor da lista se tornará um neurônio da camada subsequente. Na sequência é criada a fase de camadas densas da rede, onde a primeira camada delas possui em sua estrutura 128 neurônios e é aplicada a função de ativação relu. Subsequente é feito o processo de seleção de uma amostra desses neurônios aleatoriamente para serem descartados do processo, apenas a fim de filtrar essa camada e poupar processamento, isto é feito por meio da ferramenta Dropout, parametrizada para descartar 20% dos neurônios da camada. Finalmente é criada a camada de saída, onde haverá apenas 1 neurônio, passando pela função de ativação sigmoid em função de estarmos classificando de forma binária (ou gato ou cachorro), se fossem usadas 3 ou mais amostras lembre-se que a função de ativação seria a softmax.

```

44 classificador.compile(optimizer = 'adam',
45     loss = 'binary_crossentropy',
46     metrics = ['accuracy'])
47 classificador.fit_generator(base_treino,
48     steps_per_epoch = 4000,
49     epochs = 5,
50     validation_data = base_teste,
51     validation_steps = 1000)

```

Então é criada a estrutura de compilação da rede, nada diferente do usual usado em outros modelos e por fim é criada a camada que alimenta a rede e a coloca em execução, note que agora, em um modelo de rede neural convolucional, este processo se dá pela função .fit_generator() onde passamos como parâmetros os dados contidos em base_treino, steps_per_epoch define o

número de execuções sobre cada amostra (se não houver 4000 amostras a rede irá realizar testes sobre amostras repetidas ou geradas anteriormente), epochs como de costume define que a toda a rede neural será executada 5 vezes (reforçando seu aprendizado), validation_data irá, como o próprio nome dá a ideia, fazer a verificação para ver se os dados encontrados pela rede conferem perfeitamente com os da base de teste em estrutura e por fim é definido que essa validação será feita sobre 1000 amostras aleatórias. Selezionando e executando todo esse bloco de código, se não houve nenhum erro de sintaxe você verá a rede em execução via console.

```
Use tf.cast instead.
Epoch 1/5
4000/4000 [=====] - 3844s 961ms/step - loss: 0.4491 - 
acc: 0.7805 - val_loss: 0.5055 - val_acc: 0.7933
Epoch 2/5
4000/4000 [=====] - 6296s 2s/step - loss: 0.2024 - acc
0.9180 - val_loss: 0.7441 - val_acc: 0.7602
Epoch 3/5
4000/4000 [=====] - 3686s 921ms/step - loss: 0.1213 - 
acc: 0.9537 - val_loss: 0.7576 - val_acc: 0.7867
Epoch 4/5
4000/4000 [=====] - 65106s 16s/step - loss: 0.0889 - a
0.9668 - val_loss: 0.7722 - val_acc: 0.7861
```

Você irá notar duas coisas a partir desse modelo, primeira delas é que mesmo realizadas todas as conversões, o processamento da mesma é relativamente lento, uma vez que internamente existe um enorme volume de dados a ser processado. Segundo, que este modelo de rede neural se inicia com uma margem de acertos bastante baixa (78% na primeira época de execução) e à medida que vai avançando em épocas essa margem sobe consideravelmente (na quarta época a margem de acertos era de 96%).

```
Epoch 2/5
4000/4000 [=====] - 6296s 2s/step - loss: 0.2024 - acc:
0.9180 - val_loss: 0.7441 - val_acc: 0.7602
Epoch 3/5
4000/4000 [=====] - 3686s 921ms/step - loss: 0.1213 - 
acc: 0.9537 - val_loss: 0.7576 - val_acc: 0.7867
Epoch 4/5
4000/4000 [=====] - 65106s 16s/step - loss: 0.0889 - a
0.9668 - val_loss: 0.7722 - val_acc: 0.7861
Epoch 5/5
4000/4000 [=====] - 4074s 1s/step - loss: 0.0713 - acc:
0.9741 - val_loss: 0.8693 - val_acc: 0.7931
Traceback (most recent call last):
|
```

Por fim na última época a margem de acertos final foi de 97%, lembrando que aqui, apenas como exemplo, executamos este modelo apenas 5 vezes, em aplicações reais é possível aumentar e estabilizar essa margem ainda mais, executando a rede por mais épocas, fazendo valer mais seu aprendizado por reforço.

```
53 import numpy as np
54 from keras.preprocessing import image
55 imagem_teste = image.load_img('dataset/test_set/gato/cat.3538.jpg',
56                               target_size = (64,64))
57 imagem_teste = image.img_to_array(imagem_teste)
58 imagem_teste = np.expand_dims(imagem_teste,
59                               axis = 0)
60 previsor = classificador.predict(imagem_teste)
```

Uma vez realizada a execução de nossa rede neural artificial convolucional, temos em mãos um modelo pronto para ser usado para testes. Como de costume, para não nos estendermos muito aqui no livro, podemos realizar um simples teste sobre uma amostra, algo próximo da aplicação real justamente por uma rede dessa ser usada para classificar um animal a partir de uma imagem fornecida. Para isso realizamos a importação da numpy que até o momento não havíamos utilizado nesse modelo, também realizamos a importação da ferramenta image do módulo preprocessing da biblioteca keras. Em seguida é criada a variável imagem_teste que por meio da função image.load_img() faz a leitura de um arquivo escolhido definido pelo usuário, apenas complementando, é parametrizado que esta imagem no processo de leitura, independentemente de seu tamanho original, será convertida para 64x64.



A imagem em questão é essa, escolhida aleatoriamente (e lembrando que apesar de sua nomenclatura “cat...” o interpretador fará a leitura, as conversões e a identificação com base nas características da imagem. Na sequência é feita a conversão da imagem propriamente dita para uma array do tipo numpy por meio da função img_to_array(). Em seguida é parametrizado que expand_dims permite que tais dados sejam empilhados em forma de lista por meio do parâmetro axis = 0, lembre-se que na fase onde os dados passam pela camada Flatten, eles serão convertidos para dados sequenciais que se tornarão neurônios de uma camada da rede. Por fim, por meio da função .predict() é passada esta imagem pelo processamento de nossa rede para que seja identificada como tal.

Nome	Tipo	Tamanho	Valor
imagem_teste	float32	(1, 64, 64, 3)	[[[[37. 45. 48.] [34. 46. 62.]
previsor	float32	(1, 1)	[[1.]]
resultado_final	dict	2	{'cachorro':0, 'gato':1}

O retorno neste caso foi, como esperado, que a imagem fornecida foi de fato classificada como uma imagem de um gato.

Código Completo:

```

1 import numpy as np
2 from keras.preprocessing import image
3 from keras.models import Sequential
4 from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
5 from keras.layers.normalization import BatchNormalization
6 from keras.preprocessing.image import ImageDataGenerator
7
8 gerador_treino = ImageDataGenerator(rescale = 1.0/255,
9                                     rotation_range = 7,
10                                    horizontal_flip = True,
11                                    shear_range = 0.2,
12                                    height_shift_range = 0.07,
13                                    zoom_range = 0.2)
14 gerador_teste = ImageDataGenerator(rescale = 1.0/255)
15
16 base_treino = gerador_treino.flow_from_directory('dataset/training_set',
17                                                 target_size = (64,64),
18                                                 batch_size = 32,
19                                                 class_mode = 'binary')
20 base_teste = gerador_teste.flow_from_directory('dataset/test_set',
21                                                 target_size = (64,64),
22                                                 batch_size = 32,
23                                                 class_mode = 'binary')
24
25 classificador = Sequential()
26 classificador.add(Conv2D(32,
27                         (3,3),
28                         input_shape = (64,64,3),
29                         activation = 'relu'))
30 classificador.add(BatchNormalization())
31 classificador.add(MaxPooling2D(pool_size = (2,2)))
32 classificador.add(Conv2D(32,
33                         (3,3),
34                         activation = 'relu'))
35 classificador.add(BatchNormalization())
36 classificador.add(MaxPooling2D(pool_size = (2,2)))
37 classificador.add(Flatten())
38 classificador.add(Dense(units = 128,
39                         activation = 'relu'))
40 classificador.add(Dropout(0.2))
41 classificador.add(Dense(units = 128,
42                         activation = 'relu'))
43 classificador.add(Dropout(0.2))
44 classificador.add(Dense(units = 1,
45                         activation = 'sigmoid'))
46 classificador.compile(optimizer = 'adam',
47                       loss = 'binary_crossentropy',
48                       metrics = ['accuracy'])
49 classificador.fit_generator(base_treino,
50                             steps_per_epoch = 4000,
51                             epochs = 5,
52                             validation_data = base_teste,
53                             validation_steps = 1000)
54
55 imagem_teste = image.load_img('dataset/test_set/gato/cat.3538.jpg',
56                               target_size = (64,64))
57 imagem_teste = image.img_to_array(imagem_teste)
58 imagem_teste = np.expand_dims(imagem_teste,
59                               axis = 0)
60 previsor = classificador.predict(imagem_teste)
61 base_treino.class_indices

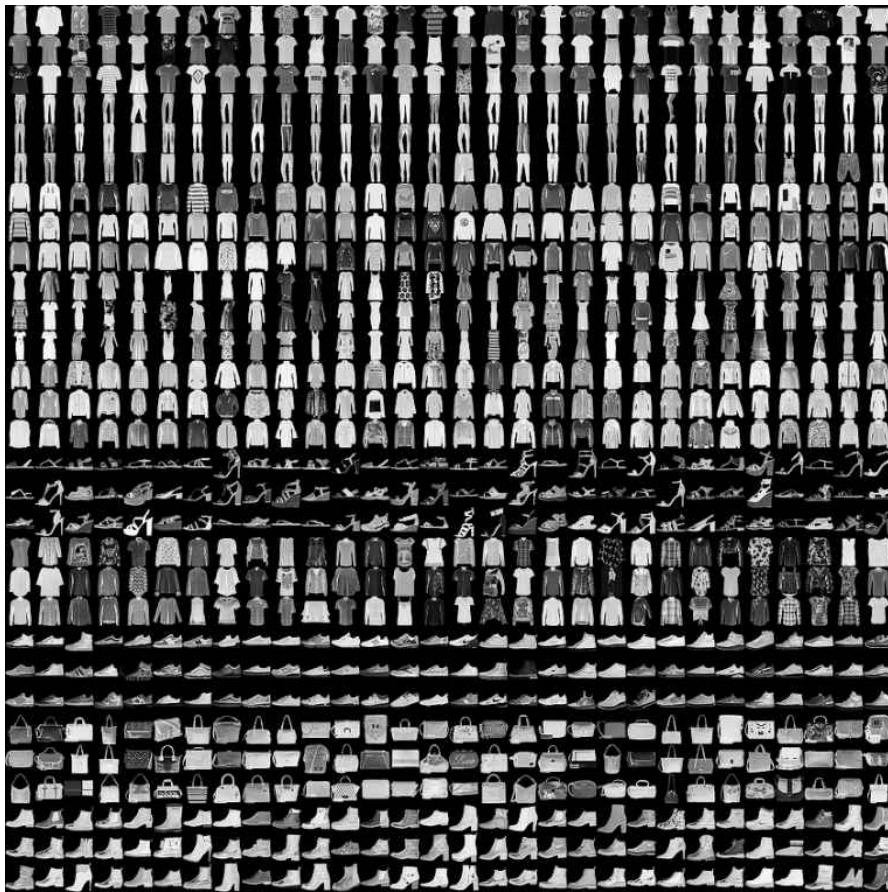
```


Classificação a Partir de Imagens – TensorFlow – Fashion Dataset

Para finalizar este capítulo, vamos dar uma breve explanada sobre como podemos criar modelos de redes neurais artificiais convolucionais a partir do TensorFlow. Esta biblioteca, desenvolvida pelo Google e liberada para uso no modelo open-source, possui um modelo lógico e estrutural que por sua vez vem sendo aprimorado pela comunidade para que tal ferramenta seja de fácil acesso e o principal, ofereça melhor performance se comparado com as outras bibliotecas científicas desenvolvidas para Python. Internamente, o modelo de funcionamento de uma rede neural via TensorFlow se difere do convencionam uma vez que sua estrutura lógica segue um padrão de fluxo, onde as camadas de processamento realizam processamento em blocos e dessa forma se obtém melhor performance.

Raciocine que em um modelo de rede neural como viemos trabalhando até então criamos a estrutura de camadas “engessadas” entre si, uma pequena alteração que impeça a comunicação correta entre camadas resulta em uma rede que não consegue ser executada. O principal do TensorFlow veio justamente nesse quesito, existe obviamente a comunicação entre camadas da rede, mas estas por sua vez podem ser desmembradas e processada a parte. De forma geral, para fácil entendimento, via TensorFlow é possível pegar um problema computacional de grande complexidade e o dividir para processamento em pequenos blocos (até mesmo realizado em máquinas diferentes) para que se obtenha maior performance.

Para nosso exemplo, usaremos de uma base de dados Fashion MNIST, base essa integrante das bases de exemplo da biblioteca keras.



Basicamente a biblioteca Fashion conta com 70000 amostras divididas em 10 categorias diferentes de roupas e acessórios. A partir dessa base, treinaremos uma rede neural capaz de classificar novas peças que forem fornecidas pelo usuário. O grande diferencial que você notará em relação ao modelo anterior é a performance. Muito provavelmente você ao executar o modelo anterior teve um tempo de execução da rede de mais ou menos uma hora. Aqui faremos um tipo de classificação com um volume considerável de amostras que será feito em poucos segundos via TensorFlow.

```
File: Fashion Dataset TensorFlow.py
```

```
1 from __future__ import absolute_import, division, print_function, unicode_literals
2 import tensorflow as tf
3 from tensorflow import keras
4 import numpy as np
5 import matplotlib.pyplot as plt
```

Como sempre o processo se inicia com a importação das bibliotecas, módulos e ferramentas necessárias. Note que aqui já de início estamos realizando um tipo de importação até agora não vista em outro modelo, `from __future__` estamos importando uma série de ferramentas internas ou experimentais do Python, a nomenclatura com “`__`” antes e depois de um determinado nome significa que tal nome é reservado para uma função interna, não sendo possível atribuir a uma variável ou outra coisa. De `__future__` importamos `absolute_import`, `division`, `print_function` e `unicode_literals`. Em seguida importamos `tensorflow` e a referenciamos como `tf`, já as linhas abaixo são de importações de bibliotecas conhecidas.

```

7 base = keras.datasets.fashion_mnist
8

```

Em seguida criamos nossa variável de nome base que recebe o conteúdo de fashion_mnist das bibliotecas exemplo do keras.

```

9 (etreino,streino),(eteste,steste) = base.load_data()
10
11 rotulos = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt',
12     'Sneaker','Bag','Ankle boot']
13
14 etreino = etreino / 255
15 streino = streino / 255

```

Após selecionada e executada a linha de código anterior, é feita a respectiva importação, na sequência podemos criar nossas variáveis designadas para treino e teste que recebem o conteúdo contido em base, importado anteriormente. Da mesma forma é criada a variável rotulos que por sua vez recebe como atributos, em forma de lista, os nomes das peças de vestuário a serem classificadas. Para finalizar este bloco, realizamos a conversão dos dados brutos de int para float para que possamos os tratar de forma matricial.

Nome	Tipo	Tamanho	Valor
eteste	uint8	(10000, 28, 28)	[[[0 0 0 ... 0 0 0] [0 0 0 ... 0 0 0]]]
etreino	float64	(60000, 28, 28)	[[[0. 0. 0. ... 0. 0. 0.] [0. 0. 0. ... 0. 0. 0.]
rotulos	list	10	['T-shirt/top', 'Trouser', 'Pullover', 'Dres 'Coat', 'Sandal', 'Shi ...
steste	uint8	(10000,)	[9 2 1 ... 8 1 5]
streino	float64	(60000,)	[0.03529412 0. 0. ... 0.0117 0. 0.01960784 ...]

Se até o momento não houve nenhum erro de sintaxe, podemos verificar via explorador de variáveis que de fato foram criadas as referentes variáveis para entradas de treino e teste assim como para as saídas de treino e teste. Apenas por hora note que etreino possui 60000 amostras que por sua vez estão no formato de uma matriz 28x28, eteste por sua vez possui 10000 amostras no mesmo formato.

```

17 plt.figure(figsize = (10,10))
18 for i in range(25):
19     plt.subplot(5,5,i+1)
20     plt.xticks([])
21     plt.yticks([])
22     plt.grid(False)
23     plt.imshow(etreino[i],
24                cmap = plt.cm.binary)
25     plt.xlabel(rotulos,[steste[i]])
26 plt.show()

```

Em seguida podemos fazer a visualização de uma amostra dessa base de dados, aqui, um pouco diferente do que estamos habituados a fazer, vamos reconstruir uma imagem pixel a pixel e exibir em nosso console. Para isso passaremos alguns argumentos para nossa matplotlib, o primeiro deles é que a imagem a ser exibida terá um tamanho de plotagem (e não de pixels) 10x10, em seguida criamos um laço de repetição para reagruparmos os dados necessários a reconstrução. Basicamente o que parametrizamos é a importação do primeiro objeto de nossa base, reagrupando todos dados referentes ao seu mapeamento de pixels e exibindo em tela.



['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

Selecionando e executando o bloco de código anterior podemos ver que a plotagem foi feita de forma correta, inclusive é possível reconhecer que tal amostra é uma bota.

```
28 classificador = keras.Sequential([
29     keras.layers.Flatten(input_shape=(28, 28)),
30     keras.layers.Dense(128, activation=tf.nn.relu),
31     keras.layers.Dense(10, activation=tf.nn.softmax)
32 ])
```

Dando início a criação da estrutura da rede neural, note algumas particularidades no que diz respeito ao formato da codificação mesmo. Inicialmente criamos nossa variável de nome classificador que imediatamente instancia e inicializa keras.Sequential(), internamente já fazemos a criação das camadas, sendo a primeira delas já realizando o processo de Flatten, pegando a amostra em tamanho 28x28, estendendo a mesma e definindo cada valor como um neurônio da camada de entrada. Logo após é criada uma camada intermediária densa onde apenas é necessário definir o número de neurônios (128) e como ativação usamos relu, porém agora a sintaxe nos mostra que estamos usando relu função interna da biblioteca TensorFlow por meio do parâmetro tf.nn.relu. Por fim é criada a camada de saída, com 10 neurônios e da mesma forma que na camada anterior, usamos a função de ativação interna do TensorFlow, neste caso, como é uma classificação com 10 saídas diferentes, softmax.

```
34 classificador.compile(optimizer='adam',
35                         loss='sparse_categorical_crossentropy',
36                         metrics=['accuracy'])
37 classificador.fit(entreino,
38                     streino,
39                     epochs = 5)
```

A estrutura da rede neural convolucional já está pronta (sim, em 3 linhas de código), podemos assim criar as camadas de compilação da mesma, com os mesmos moldes que estamos acostumados, e a camada que alimentará a rede e a colocará em funcionamento, onde apenas repassamos como dados o conteúdo de entreino e streino e definimos que tal rede será executada 5 vezes, não é necessário nenhum parâmetro adicional.

Como comentado anteriormente, você se surpreenderá com a eficiência desse modelo, aqui, processando 60000 amostras em um tempo médio de 3 segundos.

```

...
          epochs = 5)
Epoch 1/5
60000/60000 [=====] - 6s 107us/sample - loss: 0.0028 -
acc: 0.9995
Epoch 2/5
60000/60000 [=====] - 6s 102us/sample - loss: 3.7276e-6
- acc: 1.0000
Epoch 3/5
60000/60000 [=====] - 6s 101us/sample - loss: 1.7924e-6
- acc: 1.0000
Epoch 4/5
60000/60000 [=====] - 6s 102us/sample - loss: 1.2905e-6
- acc: 1.0000
Epoch 5/5
60000/60000 [=====] - 6s 102us/sample - loss: 1.1569e-6
- acc: 1.0000

```

Poucos segundos após o início da execução da rede a mesma já estará treinada, note que ela atinge uma margem de acertos de 99% (sendo honesto, o indicador de acc nos mostra 1.0 – 100%, mas este valor não deve ser real, apenas algo muito próximo a 100%)

Pronto, sua rede neural via TensorFlow já está treinada e pronta para execução de testes.

Código Completo:

```

1 from __future__ import absolute_import, division, print_function, unicode_literals
2 import tensorflow as tf
3 from tensorflow import keras
4 import numpy as np
5 import matplotlib.pyplot as plt
6
7 base = keras.datasets.fashion_mnist
8
9 (etreino,streino),(eteste,steste) = base.load_data()
10
11 rotulos = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
12             'Sneaker', 'Bag', 'Ankle boot']
13
14 etreino = etreino / 255
15 streino = streino / 255
16
17 plt.figure(figsize = (10,10))
18 for i in range(25):
19     plt.subplot(5,5,i+1)
20     plt.xticks([])
21     plt.yticks([])
22     plt.grid(False)
23     plt.imshow(etreino[i],
24                cmap = plt.cm.binary)
25     plt.xlabel(rotulos,[steste[i]])
26 plt.show()
27
28 classificador = keras.Sequential([
29     keras.layers.Flatten(input_shape=(28, 28)),
30     keras.layers.Dense(128, activation=tf.nn.relu),
31     keras.layers.Dense(10, activation=tf.nn.softmax)
32 ])
33
34 classificador.compile(optimizer='adam',
35                       loss='sparse_categorical_crossentropy',
36                       metrics=['accuracy'])
37 classificador.fit(etreino,
38                   streino,
39                   epochs = 5)

```


33 – Redes Neurais Artificiais Recorrentes

Avançando com nossos estudos, hora de dedicarmos um tempinho a entender o que são as chamadas redes neurais recorrentes. Diferente dos demais tipos vistos até agora, este modelo de rede recebe um capítulo específico para si em função de ser um modelo com certas particularidades que os modelos anteriores não possuíam.

O primeiro grande diferencial de uma rede neural recorrente é que, diferente as outras, ela é focada no processamento de dados sequenciais dentro de um intervalo de tempo, dessa forma, pós processamento da rede é possível que sejam realizadas previsões das ações seguintes que irão ocorrer dentro da escala de tempo. Para ficar mais claro, imagine um consultório médico, são captados e processados os dados de todos agendamentos de todos os dias das 8 da manhã até as 5 da tarde, digamos que o consultório passará a ter expediente até as 6 horas da tarde, a rede será capaz de descobrir o padrão das horas anteriores e prever qual será a demanda de atendimentos nessa hora adicional. O exemplo que usaremos a seguir é uma análise temporal de abertura e fechamento de valores de ações, para realizar previsão se uma determinada ação irá subir ou baixar de valor com base no padrão dos dias anteriores.

Estes tipos de redes são facilmente adaptáveis para reconhecimento de linguagem natural e até mesmo previsão de texto. Tudo isso se dá por uma característica desse tipo de rede que é a chamada memória de longo e curto intervalo de tempo.

Em inglês Long Short-term Memory, é um modelo de rede neural artificial recorrente onde a rede tem capacidade de trabalhar em loops definidos para que se encontre padrões de informações persistentes. Raciocine por hora que essa arquitetura cria um modelo onde existem informações de entrada, o algoritmo irá decidir o que será armazenado e o que será removido com base numa primeira etapa de processamento de dados sob aplicação de uma função sigmoid, se o retorno for próximo a 0 tal dado é descartada, se próximo a 1 esse dado é mantido e reprocessado no mesmo neurônio, agora sob a aplicação de uma função chamada tangente hiperbólica. Em seguida atualiza o estado antigo e parte para a etapa onde está situada a camada de saída. Dessa forma é criada uma espécie de “memória” capaz de ser reutilizada a qualquer momento para autocompletar um determinado padrão.

Previsão de Séries Temporais – Bolsa Dataset

Para entendermos de fato, e na prática, o que é uma rede neural artificial recorrente, vamos usar de uma das principais aplicações deste modelo, a de previsão de preços de ações. O modelo que estaremos criando se aplica a todo tipo de previsão com base em série temporal, em outras palavras, sempre que trabalharmos sobre uma base de dados que oferece um padrão em relação a tempo, seja horas, dias, semanas, meses, podemos fazer previsões do futuro imediato. Para isso, usaremos de uma base real e atual extraída diretamente do site Yahoo Finanças.

The screenshot shows the Yahoo Finance homepage. At the top, there's a search bar with placeholder text "Pesquise por notícias, símbolos ou empresas". Below the search bar are navigation links: Início, Finanças Pessoais, Carreira, Tecnologia, Economia, Carros, Cotações, Ações, and Meu portfólio. On the right side, there are buttons for "Entrar" and "Mail". The main content area features several market indices: BOVESPA (97.603,01, -2.408,27 (-2,41%)), Merval (27.024,46, -958,50 (-3,43%)), MXV (40.009,00, -135,24 (-0,34%)), PETROLEO CRU (53,58, -1,77 (-3,20%)), OURO (1.535,20, +26,70 (+1,77%)), and Bitcoin USD (10.418,08, +313,75 (+3,11%)). To the right of these, a message says "Mercado fechará em 3 h 6 min". Below the indices, there's a large image of a man in a suit and tie, with the headline "Morre um dos 10 homens mais ricos do mundo". Further down, there are five news thumbnails: "Polícia Federal mira BTG e dono da Caixa", "Empreendedora fatura milhões com coxinha", "Crise na Amazônia ameaça acordo UE-Mercosul", "'Brasil deveria privatizar estatais de graça'", and "Brasil criou 43.820 vagas de empregos em julho". On the right side, there are sections for "Meu portfólio e mercados", "Personalizar", "Visualizados Recentemente", "Minha Lista", and "Criptomoedas". The "Criptomoedas" section shows BTC-USD (10.418,08, +313,75, +3,11%) and Bitcoin USD.

Inicialmente acessamos o site do Yahoo Finanças, usando a própria ferramenta de busca do site pesquisamos por Petrobras.

The screenshot shows the search results for "petrobras" on the Yahoo Finance website. The search bar at the top contains the query "petrobras". Below the search bar, there are two sections: "Símbolos" and "Painel de ações e mais". The "Símbolos" section lists several Petrobras-related stocks: PBR (Petróleo Brasileiro S.A. - Petrobras), PBR-A (Petróleo Brasileiro S.A. - Petrobras), PETR4.SA (Petróleo Brasileiro S.A. - Petrobras), APBR.BA (Petróleo Brasileiro S.A. - Petrobras), PETR3.SA (Petróleo Brasileiro S.A. - Petrobras), and BRDT3.SA (Petrobras Distribuidora S.A.). The "Painel de ações e mais" section is partially visible on the right. The stock APBR.BA is highlighted with a blue border.

A base que usaremos é a PETR4.SA.

Petróleo Brasileiro S.A. - Petrobras (PETR4.SA) [Adicionar à lista](#)

24,55 -0,67 (-2,66%)

A partir de 2:54PM BRT. Mercado aberto.



Na página dedicada a Petrobras, é possível acompanhar a atualização de todos os índices.

Data	Abrir	Alto	<th>Fechamento*</th> <th>Fechamento ajustado**</th> <th>Volume</th>	Fechamento*	Fechamento ajustado**	Volume
23 de ago de 2019	24,78	25,37	24,31	24,56	24,56	43.866.200
22 de ago de 2019	25,50	25,72	25,16	25,22	25,22	40.808.800
21 de ago de 2019	24,35	25,99	24,18	25,45	25,45	87.840.800
20 de ago de 2019	23,91	24,19	23,68	24,02	24,02	37.141.700
19 de ago de 2019	24,30	24,50	23,85	24,03	24,03	50.699.300
16 de ago de 2019	24,72	24,77	23,89	23,91	23,91	56.782.000
15 de ago de 2019	24,99	25,00	24,14	24,23	24,23	53.894.500

Por fim clicando em Dados Históricos é possível definir um intervalo de tempo e fazer o download de todos os dados em um arquivo formato .csv. Repare que a base que usaremos possui como informações a data (dia), o valor de abertura da ação, o valor mais alto atingido naquele dia, assim como o valor mais baixo e seu valor de fechamento. Essas informações já são mais do que suficientes para treinar nossa rede e realizarmos previsões a partir dela. Apenas lembrando que um ponto importante de toda base de dados é seu volume de dados, o interessante para este modelo é você em outro momento baixar uma base de dados com alguns anos de informação adicional, dessa forma o aprendizado da rede chegará a melhores resultados.

Bolsa Dataset.py X

```

1 import numpy as np
2 import pandas as pd
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from sklearn.preprocessing import MinMaxScaler

```

Como sempre, o processo se inicia com as devidas importações. Para nosso modelo importamos como de costume, as bibliotecas numpy e pandas, assim como as ferramentas Sequential, Dense e Dropout de seus respectivos módulos da biblioteca keras.

```
7 base = pd.read_csv('petr4-treinamento.csv')
8
```

Em seguida, como de costume, criamos nossa variável de nome base que receberá todos os dados importados pela função `.read_csv()` de nosso arquivo baixado do site Yahoo Finanças.

```
9 base = base.dropna()
10 base_treino = base.iloc[:, 1:2].values
```

Na sequência realizamos um breve polimento dos dados, primeiro, excluindo todos registros faltantes de nossa base via função `.dropna()`, em seguida criamos nossa variável de nome `base_treino` que recebe pelo método `.iloc[].values` todos os dados de todas as linhas, e os dados da coluna 1 (lembrando que fora a indexação criada pela IDE, a leitura de todo arquivo `.csv` começa em índice 0), referente aos valores de abertura das ações.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, High, Low, Close, A Close, Volume
base_treino	float64	(1242, 1)	[[19.99] [19.809999]]

Selecionando e executando o bloco de código anterior podemos verificar que as variáveis foram criadas corretamente, delas, `base_treino` agora possui 1242 registros divididos em apenas uma categoria (valor de abertura da ação).

```
12 normalizador = MinMaxScaler(feature_range = (0,1))
13 base_treino_normalizada = normalizador.fit_transform(base_treino)
```

Em seguida realizaremos uma etapa de polimento desses dados bastante comum a este tipo de rede neural, onde normalizaremos os dados dentro de um intervalo numérico que fique mais fácil a visualização dos mesmos assim como a aplicação de certas funções. Para isso criamos uma variável de nome `normalizador` que inicializa a ferramenta `MinMaxScaler`, nela, passamos como parâmetro `feature_range = (0,1)`, o que em outras palavras significa que por meio dessa ferramenta transformaremos nossos dados em um intervalo entre 0 e 1, números entre 0 e 1. Logo após criamos nossa variável `base_treino_normalizada` que aplica essa transformação sobre os dados de `base_treino` por meio da função `.fit_transform()`.

base_treino_normalizada - Matriz NumPy	
0	0.765019
1	0.756298
2	0.781492
3	0.78876
4	0.770833
5	0.748062

Cor de fundo

Selecionado e executado o bloco de código anterior podemos visualizar agora nossa base de dados de treino normalizados como números do tipo float32, categorizados em um intervalo entre 0 e 1.

```

15 previsores = []
16 preco_real = []
17
18 for i in range(90,1242):
19     previsores.append(base_treino_normalizada[i-90:i,0])
20     preco_real.append(base_treino_normalizada[i,0])
21
22 previsores, preco_real = np.array(previsores), np.array(preco_real)
23 previsores = np.reshape(previsores,(previsores.shape[0], previsores.shape[1],1)

```

Uma vez terminado o polimento de nossos dados de entrada, podemos realizar as devidas separações e conversões para nossos dados de previsão e de saída, que faremos a comparação posteriormente para avaliação da performance do modelo. Inicialmente criamos as variáveis previsores e preco_real com listas vazias como atributo. Em seguida criamos um laço de repetição que percorrerá nossa variável base_treino_normalizada separando os primeiros 90 registros (do número 0 ao 89) para que a partir dela seja encontrado o padrão que usaremos para nossas previsões. Por fim, ainda dentro desse laço, pegamos os registros de nº 90 para usarmos como parâmetro, como preço real a ser prevista pela rede. Fora do laço, realizamos as conversões de formato, transformando tudo em array do tipo numpy e definindo que a posição 0 na lista de previsores receberá os 1152 registros, enquanto a posição 1 receberá as informações referentes aos 90 dias usados como previsão.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, High, Low, Close, Adj Close, Volume
base_treino	float64	(1242, 1)	[[19.99] [19.809999]]
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]]
i	int	1	1241
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.7747]
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]]

Uma vez encerrado o processo de tratamento dos dados, é possível conferir cada variável assim como seus valores via explorador de variáveis, e mais importante que isso, agora podemos dar início a criação do nosso modelo de rede neural recorrente.

```
25 from keras.layers import LSTM
26
```

Como de praxe, não será necessário criar manualmente toda a estrutura de nossa rede neural, mas criar a partir de uma ferramenta que possui essa estrutura pré-pronta e configurada. Par isso, realizamos a importação da ferramenta LSTM do módulo layers da biblioteca keras.

```
27 regressor = Sequential()
28 regressor.add(LSTM(units = 100,
29                     return_sequences = True,
30                     input_shape = (previsores.shape[1],1)))
31 regressor.add(Dropout(0.3))
32 regressor.add(LSTM(units = 50,
33                     return_sequences = True,))
34 regressor.add(Dropout(0.3))
35 regressor.add(LSTM(units = 50))
36 regressor.add(Dropout(0.3))
37 regressor.add(Dense(units = 1,
38                     activation = 'linear'))
```

Dando início a codificação da estrutura da rede, criamos inicialmente nossa variável regressor que inicializa Sequential sem parâmetros. Na sequência adicionamos a camada LSTM, que por sua vez tem como parâmetros input_shape que recebe os dados referentes aos registros e intervalo de tempo de nossa variável previsores, retornando essa sequência (recorrência) via parâmetro return_sequences definido como True (False será sem recorrência, em outras palavras, sem atualização de dados no próprio neurônio da camada) e por fim units especificando quantos neurônios estarão na primeira camada oculta. Em seguida é adicionada a primeira camada de Dropout, que por sua vez, descartará 30% dos neurônios dessa camada aleatoriamente. Logo após criamos mais uma camada LSTM onde há recorrência, agora com tamanho de 50 neurônios. A seguir é adicionada mais uma camada Dropout, descartando mais 30% dos neurônios da camada, mais uma camada LSTM agora sem recorrência, ou seja, os valores que preencherem os neurônios dessa camada agora são estáticos, servindo apenas como referência para camada subsequente. Por fim é criada uma camada Dense, onde há apenas 1 neurônio de saída e seu método de ativação é ‘linear’, que em outras palavras é uma função de ativação nula, o valor que chegar até esse neurônio simplesmente será replicado.

```

39 regressor.compile(optimizer = 'rmsprop',
40                     loss = 'mean_squared_error',
41                     metrics = ['mean_absolute_error'])
42 regressor.fit(previsores,
43                 preco_real,
44                 epochs = 100,
45                 batch_size = 32)

```

Como de costume, criadas as camadas estruturais da rede, hora de criar a camada de compilação da mesma e a que a alimentará, colocando a rede em execução. Sendo assim criamos nossa camada de compilação onde como parâmetros temos optimizer = ‘rmsprop’, método de otimização muito usado nesses tipos de modelo em função de oferecer uma descida de gradiente bastante precisa quando trabalhada uma série temporal, loss definimos como ‘mean_squared_error’, já usada em outro modelo, onde os valores de erro são elevados ao quadrado, dando mais peso aos erros e buscando assim melhor precisão, por fim metrics é parametrizado com nosso já conhecido ‘mean_absolute_error’. Para alimentação da rede via camada .fit() passamos todos os dados contidos em previsores, preco_real, assim como definimos que a rede será executada 100 vezes, atualizando seus pesos de 32 em 32 registros.

```

mean_absolute_error: 0.0288
Epoch 97/100
1152/1152 [=====] - 8s 7ms/step - loss: 0.0013 -
mean_absolute_error: 0.0275
Epoch 98/100
1152/1152 [=====] - 8s 7ms/step - loss: 0.0013 -
mean_absolute_error: 0.0267
Epoch 99/100
1152/1152 [=====] - 8s 7ms/step - loss: 0.0014 -
mean_absolute_error: 0.0275
Epoch 100/100
1152/1152 [=====] - 8s 7ms/step - loss: 0.0014 -
mean_absolute_error: 0.0273
Out[ ]:

```

Selecionado e executado o bloco de código anterior veremos a rede sendo processada, após o término da mesma podemos ver que em suas épocas finais, graças ao fator de dar mais peso aos erros, a mesma terminou com uma margem de erro próxima a zero.

```

48 base_teste = pd.read_csv('petr4-teste.csv')
49 preco_real_teste = base_teste.iloc[:, 1:2].values
50 base_completa = pd.concat((base['Open'], base_teste['Open']), axis = 0)

```

Rede treinada hora de começarmos a realizar as devidas previsões a partir da mesma, no caso, a previsão do preço de uma ação. Para isso, inicialmente vamos realizar alguns procedimentos para nossa base de dados. Primeiramente criamos uma variável de nome base_teste que recebe todo conteúdo importado de nosso arquivo petr4-teste.csv. Da mesma forma criamos nossa variável preco_real_teste que recebe todos os valores de todas as linhas e da coluna 1 de base_teste via método .iloc[].values. Por fim criamos a variável base_completa que realiza a junção dos dados da coluna ‘Open’ de base e base teste por meio da função .concat().

Nome	Tipo	Tamanho	Valor
base	DataFrame	(1242, 7)	Column names: Date, Open, High, Low, Close, Adj Close, Volume
base_completa	Series	(1264,)	Series object of pandas.core.series module
base_teste	DataFrame	(22, 7)	Column names: Date, Open, High, Low, Close, Adj Close, Volume
base_treino	float64	(1242, 1)	[[19.99], [19.809999]]
base_treino_normalizada	float64	(1242, 1)	[[0.76501938], [0.7562984]]
i	int	1	1241
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.7747]
preco_real_teste	float64	(22, 1)	[[16.190001], [16.49]]

Via explorador de variáveis é possível visualizar todos dados de todas variáveis que temos até o momento.

```
52 entradas = base_completa[len(base_completa) - len(base_teste) - 90: ].values
53 entradas = entradas.reshape(-1,1)
54 entradas = normalizador.transform(entradas)
```

Dando continuidade, criamos uma variável de nome entradas que de forma parecida com a que fizemos anteriormente dentro de nosso laço de repetição i, recebe como atributos os últimos 90 registros (que serão usados para encontrar o padrão para previsão), em seguida são feitas as transformações de formato par cruzamento de dados. Nada muito diferente do que já fizemos algumas vezes em outros modelos.

```
56 previsores_teste = []
57 for j in range(90,112):
58     previsores_teste.append(entradas[j-90:j, 0])
59 previsores_teste = np.array(previsores_teste)
60 previsores_teste = np.reshape(previsores_teste,
61                             (previsores_teste.shape[0],
62                              previsores_teste.shape[1],
63                              1))
64 previsores_teste = regressor.predict(previsores_teste)
65 previsores_teste = normalizador.inverse_transform(previsores_teste)
```

Para criação de nossa estrutura previsora teríamos diversas opções de estrutura, seguindo como fizemos em outros modelos, vamos nos ater ao método mais simples e eficiente. Para uma série temporal, primeiro criamos uma variável de nome previsores_teste que recebe uma lista vazia como atributo, em seguida criamos um laço de repetição que irá percorrer, selecionar e copiar para previsores_teste os 90 primeiros registros de entradas (do 0 ao 89) através da função .append(). Na sequência realizamos as devidas transformações para array numpy e no formato que precisamos para haver compatibilidade de cruzamento. Por fim realizamos as previsões como de costume, usando a função .predict(). Repare na última linha deste bloco, anteriormente havíamos feito a normalização dos números para um intervalo entre 0 e 1, agora após realizadas as devidas previsões, por meio da função .inverse_transform() transformaremos novamente os dados entre 0 e 1 para seu formato original, de preço de ações.

Nome	Tipo	Tamanho	Valor
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]
entradas	float64	(112, 1)	[[0.47141473] [0.46317829]
i	int	1	1241
j	int	1	111
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.774]
preco_real_teste	float64	(22, 1)	[[16.190001] [16.49]
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]
previsores_teste	float32	(22, 1)	[[0.55642956] [0.5593487]

Repare que previsores_teste aqui está com valores normalizados.

Nome	Tipo	Tamanho	Valor
base_treino_normalizada	float64	(1242, 1)	[[0.76501938] [0.7562984]
entradas	float64	(112, 1)	[[0.47141473] [0.46317829]
i	int	1	1241
j	int	1	111
preco_real	float64	(1152,)	[0.76114341 0.76114341 0.774]
preco_real_teste	float64	(22, 1)	[[16.190001] [16.49]
previsores	float64	(1152, 90, 1)	[[[0.76501938] [0.7562984]
previsores_teste	float32	(22, 1)	[[15.684706] [15.744957]]

Após a aplicação da função `.inverse_transform()` os dados de previsores_teste voltam a ser os valores em formato de preço de ação.

```
67 previsores_teste.mean()
68 preco_real_teste.mean()
```

Como já realizado em outros modelos, aqui também se aplica fazer o uso de média dos valores, por meio da função `.mean()` tanto dos valores previstos quanto dos valores reais.

```
In [ ]: previsores_teste.mean()
Out[ ]: 17.157213

In [ ]: preco_real_teste.mean()
Out[ ]: 17.87454563636364
```

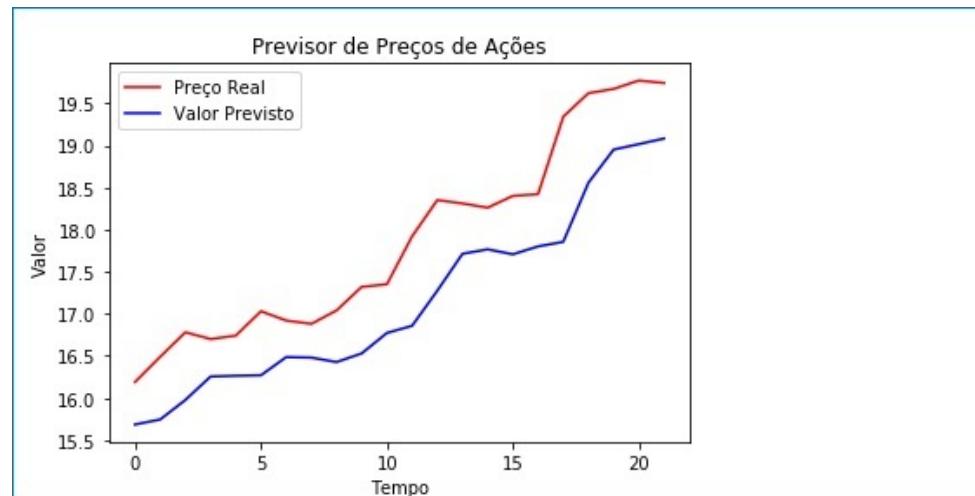
Selecionado e executado o bloco de código anterior podemos ver via console os valores retornados, note que o valor real da ação era de R\$ 17,87, enquanto o valor previsto pela rede foi de R\$ 17,15, algo muito próximo, confirmando a integridade de nossos dados assim como a eficiência de nossa rede.

Já que estamos trabalhando em cima de um exemplo de ações, quando vemos qualquer aplicação deste tipo de atividade costumamos ver percentuais ou gráficos apresentados para uma

visualização mais clara, aqui podemos fazer a plotagem de nossos resultados para ter um retorno mais interessante.

```
71 import matplotlib.pyplot as plt
72 plt.plot(preco_real_teste,
73           color = 'red',
74           label = 'Preço Real')
75 plt.plot(previsores_teste,
76           color = 'blue',
77           label = 'Valor Previsto')
78 plt.title('Previsor de Preços de Ações')
79 plt.xlabel('Tempo')
80 plt.ylabel('Valor')
81 plt.legend()
82 plt.show()
```

Como de praxe, para plotagem de nossas informações em forma de gráfico importamos a biblioteca matplotlib. De forma bastante simples, por meio da função `.plot()` passamos como parâmetro os dados de `preco_real_teste` e de `previsores_teste`, com cores diferentes para fácil visualização assim como um rótulo para cada um. Também definimos um título para nosso gráfico assim como rótulos para os planos X e Y, respectivamente. Por fim por meio da função `.show()` o gráfico é exibido no console.



Agora podemos visualizar de forma bastante clara a apresentação de nossos dados, assim como entender o padrão. Note a semelhança entre a escala dos valores reais em decorrência do tempo e dos dados puramente previstos pela rede neural. Podemos deduzir que esse modelo é bastante confiável para este tipo de aplicação.

Código Completo:

```

1 import numpy as np
2 import pandas as pd
3 from keras.models import Sequential
4 from keras.layers import Dense, Dropout
5 from sklearn.preprocessing import MinMaxScaler
6 from keras.layers import LSTM
7 import matplotlib.pyplot as plt
8
9 base = pd.read_csv('petr4-treinamento.csv')
10 base = base.dropna()
11 base_treino = base.iloc[:, 1:2].values
12 normalizador = MinMaxScaler(feature_range = (0,1))
13 base_treino_normalizada = normalizador.fit_transform(base_treino)
14
15 previsores = []
16 preco_real = []
17
18 for i in range(90,1242):
19     previsores.append(base_treino_normalizada[i-90:i,0])
20     preco_real.append(base_treino_normalizada[i,0])
21
22 previsores, preco_real = np.array(previsores), np.array(preco_real)
23 previsores = np.reshape(previsores,(previsores.shape[0], previsores.shape[1],1))
24
25 regressor = Sequential()
26 regressor.add(LSTM(units = 100,
27                     return_sequences = True,
28                     input_shape = (previsores.shape[1],1)))
29 regressor.add(Dropout(0.3))
30 regressor.add(LSTM(units = 50,
31                     return_sequences = True,))
32 regressor.add(Dropout(0.3))
33 regressor.add(LSTM(units = 50))
34 regressor.add(Dropout(0.3))
35 regressor.add(Dense(units = 1,
36                      activation = 'linear'))
37 regressor.compile(optimizer = 'rmsprop',
38                     loss = 'mean_squared_error',
39                     metrics = ['mean_absolute_error'])
40 regressor.fit(previsores,
41                 preco_real,
42                 epochs = 100,
43                 batch_size = 32)
44
45 base_teste = pd.read_csv('petr4-teste.csv')
46 preco_real_teste = base_teste.iloc[:, 1:2].values
47 base_completa = pd.concat((base['Open'], base_teste['Open']), axis = 0)
48
49 entradas = base_completa[len(base_completa) - len(base_teste) - 90: ].values
50 entradas = entradas.reshape(-1,1)
51 entradas = normalizador.transform(entradas)
52
53 previsores_teste = []
54 for j in range(90,112):
55     previsores_teste.append(entradas[j-90:j, 0])
56 previsores_teste = np.array(previsores_teste)
57 previsores_teste = np.reshape(previsores_teste,
58                               (previsores_teste.shape[0],
59                                previsores_teste.shape[1],
60                                1))
61 previsores_teste = regressor.predict(previsores_teste)
62 previsores_teste = normalizador.inverse_transform(previsores_teste)
63 previsores_teste.mean()
64 preco_real_teste.mean()

```

```
66 plt.plot(preco_real_teste,
67           color = 'red',
68           label = 'Preço Real')
69 plt.plot(previsores_teste,
70           color = 'blue',
71           label = 'Valor Previsto')
72 plt.title('Previsor de Preços de Ações')
73 plt.xlabel('Tempo')
74 plt.ylabel('Valor')
75 plt.legend()
76 plt.show()
```

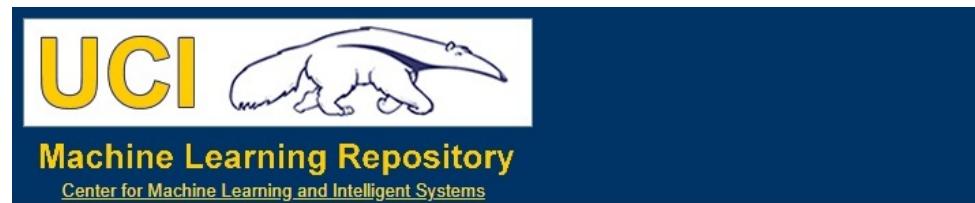
34 – Outros Modelos de Redes Neurais Artificiais

Nos capítulos anteriores percorremos uma boa parte do que temos de modelos de redes neurais. Classificadores, regressores, identificadores, previsores, etc... cada modelo com suas pequenas particularidades, mas todos integrando um contexto parecido. Agora vamos dedicar um pequeno tempo de nossos estudos a modelos de rede que não se encaixam nos moldes anteriores devido suas adaptações para resolução de problemas computacionais bastante específicos.

Mapas Auto Organizáveis – Kmeans – Vinhos Dataset

Um dos modelos mais simples dedicado ao agrupamento de objetos de mesmo tipo é o chamado modelo de mapa auto organizável. Você notará uma semelhança conceitual com o modelo estatístico KNN visto lá no início do livro, porém aqui a rede usa de alguns mecanismos para automaticamente reconhecer padrões e realizar o agrupamento de objetos inteiros (de várias características). Em algumas literaturas este modelo também é chamado Kmeans, onde existe um referencial k e com base nele a rede reconhece padrões dos objetos próximos, trazendo para o agrupamento se estes objetos possuírem características em comum, e descartando desse agrupamento objetos que não compartilham das mesmas características, dessa forma, após um tempo de execução da rede os objetos estarão separados e agrupados de acordo com sua semelhança.

Para entendermos na prática como esse modelo funciona usaremos uma base de dados disponível na UCI onde teremos de classificar amostras de vinho quanto sua semelhança, a base foi construída com base em 178 amostras, de 3 cultivadores diferentes, onde temos 13 características para definição de cada tipo de vinho, sendo assim, nossa rede irá processar tais características, identificar e agrupar os vinhos que forem parecidos.



Wine Data Set

[Download](#) [Data Folder](#) [Data Set Description](#)

Abstract: Using chemical analysis determine the origin of wines



Data Set Characteristics:	Multivariate	Number of Instances:	178	Area:	Physical
Attribute Characteristics:	Integer, Real	Number of Attributes:	13	Date Donated	1991-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	1209764

Feito o download da base de dados podemos partir diretamente para prática.

A screenshot of a code editor window titled 'Vinhos.py'. The code shown is:

```
1 import pandas as pd
2 from minisom import MiniSom
```

Como sempre, o processo se inicia com as importações dos módulos, bibliotecas e ferramentas que usaremos. Dessa forma importamos a biblioteca pandas e MiniSom da biblioteca minisom.

```
4 base = pd.read_csv('wines.csv')
5 entradas = base.iloc[:, 1:14].values
6 saidas = base.iloc[:, 0].values
```

Em seguida criamos nossas variáveis de base, entradas e saídas, importando todo conteúdo de wines.csv e dividindo essa base em atributos previsores e de saída respectivamente. O processo

segue os mesmos moldes já utilizados anteriormente em outros modelos.

Nome	Tipo	Tamanho	Valor
base	DataFrame	(178, 14)	Column names: Class, Alcohol, Malic acid, Ash, Alcalinity of ash, Ma ...
entradas	float64	(178, 13)	[[1.423e+01 1.710e+00 2.430e+00 ... 1.040e+00 3. [1 ...
saidas	int64	(178,)	[1 1 1 ... 3 3 3]

Se o processo de importação ocorreu normalmente temos à disposição as variáveis para visualização via explorador de variáveis.

```
8 from sklearn.preprocessing import MinMaxScaler  
9
```

Logo após realizamos a importação da ferramenta MinMaxScaler, do módulo preprocessing da biblioteca sklearn. Lembrando que essa ferramenta é usada para conversão de números para um intervalo definido entre um número e outro (ex: de 0 até 1).

```
10 normalizador = MinMaxScaler(feature_range = (0,1))  
11 entradas = normalizador.fit_transform(entradas)
```

Na sequência é realizado o processo de normalização nos mesmos moldes de sempre, por meio de uma variável dedicada a inicializar a ferramenta e definir um intervalo, nesse caso entre 0 e 1 mesmo, em seguida atualizando os dados da própria variável entradas.



Selecionado o bloco de código anterior e executado, podemos ver que de fato agora todos valores de entradas pertencem a um intervalo entre 0 e 1, pode parecer besteira, mas nessa escala temos um grande ganho de performance em processamento da rede.

```

13 som = MiniSom(x = 8,
14                 y = 8,
15                 input_len = 13,
16                 sigma = 1.0,
17                 learning_rate = 0.5,
18                 random_seed = 2)
19 som.random_weights_init(entradas)
20 som.train_random(data = entradas,
21                   num_iteration = 100)
22 som._weights
23 som._activation_map
24 agrupador = som.activation_response(entradas)

```

Apenas um adendo, nosso agrupador MiniSom possui uma lógica de funcionamento que você não encontrará facilmente em outras literaturas. Quando estamos falando de um mecanismo agrupador, é fundamental entender o tamanho do mesmo, uma vez que ele aplicará uma série de processos internos para realizar suas funções de identificação e agrupamento. Para definirmos o tamanho da matriz a ser usada na ferramenta devemos fazer um simples cálculo. $5 * \sqrt{n}$, onde n é o número de entradas. Nesse nosso exemplo, temos 178 entradas, dessa forma, $5 * \sqrt{178}$ resulta $5 * 13.11 = 65$ células, arredondando esse valor para um número par 64, podemos criar uma estrutura de matriz SOM de 8x8.

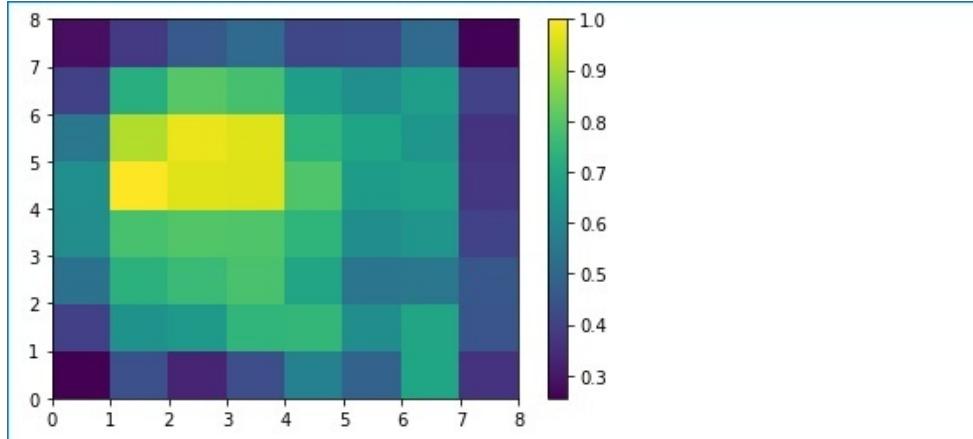
Uma vez feito o tratamento de nossos dados hora de criar a estrutura de nosso agrupador. Como estaremos usando uma ferramenta pré-pronta e pré-configurada, tudo o que temos de fazer é definir os parâmetros de tamanho, processamento e alimentação da ferramenta. Inicialmente criamos uma variável som que inicializa MiniSom passando alguns parâmetros, x e y definem o tamanho da matriz de agrupamento, input_len define quantas amostras serão usadas na entrada, sigma define a distância de proximidade entre as características das amostras, learning_rate parametriza o modo como serão atualizados os pesos e random_seed por sua vez define como se dará a alimentação da matriz para que sejam realizados os testes de proximidade, dessa vez atualizando as amostras. Na sequência são inicializados pesos com números aleatórios para entradas por meio da função .random_weights_init(). Em seguida é aplicada a função .train_random() que recebe como parâmetro os dados de entradas enquanto num_iterations define quantos testes de proximidade serão realizados entre as amostras, para cada amostra, a fim de encontrar os padrões de similaridade para agrupamento das mesmas. Para finalizar o modelo é inicializado _weights e _activation_map, finalizando criamos a variável agrupador que por meio da função .activation_response() parametrizado com entradas, criará nossa matriz de dados de agrupamento.

```

26 from pylab import pcolor, colorbar
27
28 pcolor(som.distance_map().T)
29 colorbar()

```

Por fim, para exibir nossa matriz de agrupamento de forma visual, importamos as ferramentas pcolor e colorbar da biblioteca pylab. Executando a ferramenta pcolor, parametrizada com som.distance_map().T teremos uma representação visual do agrupamento. Executando colorbar() teremos em anexo uma barra de cores que auxiliará na identificação dos tipos de vinho quanto sua proximidade.



Selecionado e executado o bloco de código anterior podemos finalmente ter uma representação visual de nossos dados, identificando facilmente por exemplo que os blocos pintados em amarelo são de um determinado tipo de vinho, diferente dos demais, assim como os outros agrupamentos de acordo com a cor.

Código Completo:

```

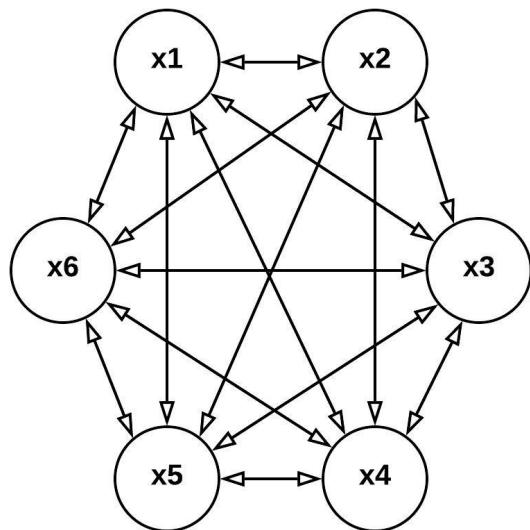
1 import pandas as pd
2 from minisom import MiniSom
3 from sklearn.preprocessing import MinMaxScaler
4 from pylab import pcolor, colorbar
5
6 base = pd.read_csv('wines.csv')
7 entradas = base.iloc[:, 1:14].values
8 saidas = base.iloc[:, 0].values
9
10 normalizador = MinMaxScaler(feature_range = (0,1))
11 entradas = normalizador.fit_transform(entradas)
12
13 som = MiniSom(x = 8,
14                 y = 8,
15                 input_len = 13,
16                 sigma = 1.0,
17                 learning_rate = 0.5,
18                 random_seed = 2)
19 som.random_weights_init(entradas)
20 som.train_random(data = entradas,
21                   num_iteration = 100)
22 som._weights
23 som._activation_map
24 agrupador = som.activation_response(entradas)
25
26 pcolor(som.distance_map().T)
27 colorbar()

```


Sistemas de Recomendação – Boltzmann Machines

Boltzmann Machines são modelos de redes neurais que são comumente usadas em sistemas de recomendação. Em algum momento você já deve ter terminado de assistir um vídeo no YouTube, Netflix ou até mesmo no Facebook e se deparou ao final do vídeo com uma série de sugestões apresentadas com base no vídeo que você acabou de assistir, muitas vezes essas recomendações parecem não fazer sentido, mas em boa parte das vezes ela realmente recomenda algo de relevância com base na sua última experiência, isto se dá por um modelo de rede neural bem característico, capaz de aprender os padrões de escolha do usuário e fazer recomendações.

Uma máquina de recomendação possui algumas particularidades, o primeiro grande destaque é o fato de as células de entrada, nesse tipo de rede, fazer parte dos nós da rede, se atualizando junto com o processamento da mesma, pois não há uma camada de entrada definida como nos outros modelos, da mesma forma não existe uma camada específica de saída. Em outros modelos havia uma fase supervisionada, onde apontávamos para a rede onde estava o erro para que ela aprendesse, neste modelo de rede neural não há supervisionamento. Dessa forma, raciocine que uma Boltzmann Machine é um tipo de rede onde todos neurônios estão conectados entre si, e não por camadas, assim a rede identifica e aprende qual é o comportamento padrão de um sistema para que possa identificar alterações, ou identificar qualquer coisa fora do padrão.



Em suma, uma Boltzmann Machine é um modelo que quando instanciado em uma determinada plataforma ou contexto, imediatamente irá identificar os objetos desse contexto (assim como suas características e classificações) e a partir daí ela reforça os padrões aprendidos e monitora qualquer padrão desconhecido.

Como exemplo vamos simular um sistema de recomendação de filme. Supondo que você é usuário de uma plataforma de streaming que ao final da visualização de um vídeo pede que você avalie o mesmo simplesmente dizendo se gostou ou se não gostou, com base nessas respostas o sistema de recomendação irá aprender que tipo de vídeo você gosta para que posteriormente possa fazer recomendações.

```

1 from rbm import RBM
2 import numpy as np

```

Dando inicio criamos nosso arquivo Boltzmann.py e em seguida já realizamos as primeiras importações. Importamos a biblioteca numpy e a ferramenta RBM (Restricted Boltzmann Machine). Para que esse processo ocorra de forma correta é necessário o arquivo rbm.py estar na mesma pasta de Boltzmann.py.

```

4 rbm = RBM(num_visible = 6,
5             num_hidden = 2)

```

Em seguida criamos nossa variável rbm que inicializa a ferramenta RBM lhe passando como parâmetros num_visible referente ao número de nós dessa rede (lembrando que aqui não são tratados como neurônios de uma camada de entrada...) e num_hidden que define o número de nós na camada oculta.

```

7 base = np.array([[1,1,1,0,0,0],
8                  [1,0,1,0,0,0],
9                  [1,1,1,0,0,0],
.0
.1
.2
[0,0,1,1,1,1],
[0,0,1,1,0,1],
[0,0,1,0,1,0]])

```

Na sequência criamos nossa variável base que tem como atributo uma array numpy onde estamos simulando 6 usuários diferentes (cada linha) e que filmes assistiu ou deixou de assistir respectivamente representado em 1 ou 0.

Nome	Tipo	Tamanho	Valor
base	int32	(6, 6)	[[1 1 1 0 0 0] [1 0 1 0 0 0]]

Se o processo foi feito corretamente, sem erros de sintaxe, a devida variável é criada.

```

14 filmes = ['O Exorcista',
15           'American Pie',
16           'Matrix',
17           'Forrest Gump',
18           'Documentário X',
19           'O Rei Leão']

```

Logo após é criada nossa variável filmes que recebe em formato de lista o nome de 6 filmes, aqui neste exemplo, apenas diferenciados quanto ao seu estilo. Um de terror, um de comédia, um de ficção científica, um drama, um documentário e uma animação.

```

21 rbm.train(base,
22             max_epochs = 3000)
23 rbm.weights

```

Na sequência executamos duas funções de nossa rbm, .train() parametrizado com os dados de nossa base e max_epochs que define quantas vezes ocorrerão as atualizações dos pesos. Por fim são inicializados os pesos.

```
Epoch 2990: error is 1.1867429671954914
Epoch 2991: error is 1.1867275151709342
Epoch 2992: error is 2.4910594588085955
Epoch 2993: error is 1.187052233844001
Epoch 2994: error is 1.18702346961957
Epoch 2995: error is 1.18699579300992
Epoch 2996: error is 1.1869691220316294
Epoch 2997: error is 1.7041098896921478
Epoch 2998: error is 1.1870667506429766
Epoch 2999: error is 1.1870341673638922
Out[ ]:
```

Selecionado e executado o bloco de código anterior é possível ver a rede sendo treinada via console assim como a margem de erro calculada.

```
array([[ 4.51019147,  1.28650412, -1.01177789],
       [-3.95724928,  1.26896524,  6.97861864],
       [-4.13637439,  0.68705828,  4.1564823 ],
       [ 6.83260098,  2.50333604,  0.04463753],
       [ 3.32804508, -6.63982934, -3.58508583],
       [ 0.02403081,  2.49566332, -6.73135502],
       [ 3.32697787, -6.63173354, -3.61145216]])
```

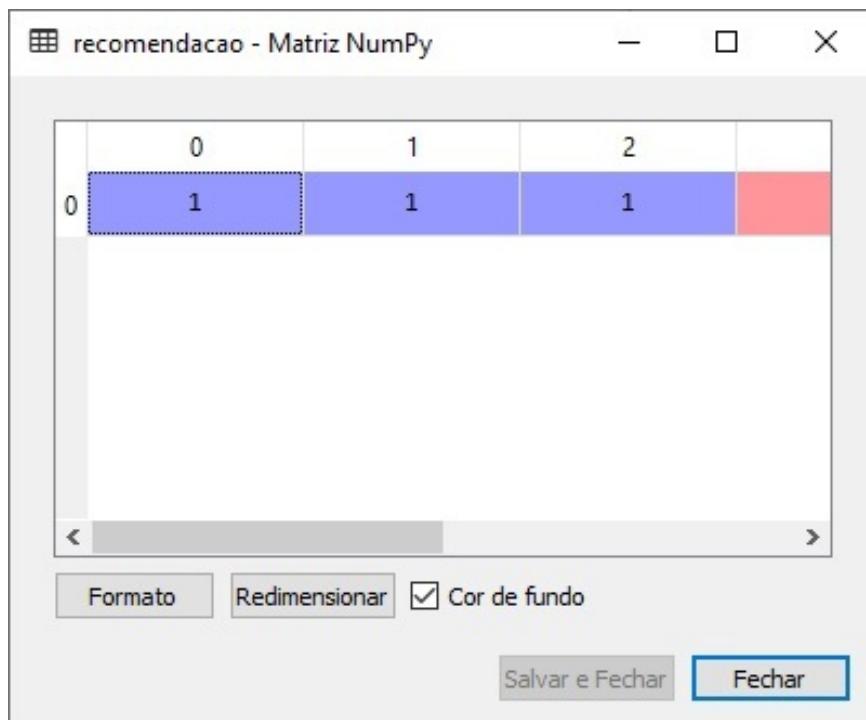
Da mesma forma é criada uma array com os dados dos pesos. Aqui a maneira correta de fazer a leitura dessa matriz é não considerar nem a primeira linha nem a primeira coluna pois esses são dados de bias, a leitura é feita a partir da segunda linha da segunda coluna em diante. Dessa forma a rbm encontrará os padrões que dão característica a cada tipo de filme, por exemplo na segunda linha, referente ao filme O Exorcista, 1.26 e 6.97. A partir desse ponto, podemos simular um usuário e fazer a recomendação do próximo filme ao mesmo.

```
26 usuario = np.array([[1,1,0,1,0,0]])
27
```

Começando com a fase de previsão e recomendação, inicialmente criamos uma variável de nome usuario, que por sua vez, com base em sua array, assistiu O Exorcista, American Pie e Forrest Gump, deixando de assistir outros títulos (essa array em outro contexto também pode significar que o usuário assistiu todos os 6 filmes mas gostou apenas do nº 1, 2 e 4).

```
28 rbm.run_visible(usuario)
29 camada_oculta = np.array([[0,1]])
30 recomendacao = rbm.run_hidden(camada_oculta)
```

Em seguida criamos o mecanismo que fará a identificação do usuário (suas preferências). Para isso usamos a função .run_visible() parametrizada com os dados de usuario. Na sequência é criada a variável camada_oculta que possui uma array para classificação binária, ou um tipo de filme ou outro com base na proximidade de suas características. Por fim é criada a variável recomendacao que roda a função .run_hidden() sobre camada_oculta. Desta maneira será identificado e recomendado um próximo filme com base nas preferências do usuário. Lembrando que aqui temos uma base de dados muito pequena, o ideal seria um catálogo de vários títulos para que se diferenciasse com maior precisão os estilos de filme.



Note que executado o bloco de código anterior, é criada a variável `recomendacao` onde podemos ver que há o registro do primeiro e do segundo filme, ao lado agora está ativado o registro correspondente ao terceiro filme, como o usuário ainda não viu este, essa coluna agora marcada como 1 será a referência para sugestão desse filme.

```
32 for i in range(len(usuario[0])):
33     print(usuario[0,i])
34     if usuario[0,i] == 0 and recomendacao[0,i] == 1:
35         print(filmes[i])
```

Para deixar mais claro esse processo, criamos um laço de repetição que percorre todos os registros de usuário (toda a linha), com base nisso é impresso o padrão das preferências desse usuário. Em seguida é criada uma estrutura condicional que basicamente verifica que se os registros do usuário forem igual a 0 e os registros de `recomendacao` na mesma posição da lista `for` igual a 1, é impresso o nome do filme dessa posição da lista, como recomendação.

Nome	Tipo	Tamanho	Valor
base	int32	(6, 6)	[[1 1 1 0 0 0] [1 0 1 0 0 0]]
camada_oculta	int32	(1, 2)	[[0 1]]
filmes	list	6	['O Exorcista', 'American Pie', 'Matrix', 'Forrest Gump', 'Documentário ...']
i	int	1	5
recomendacao	float64	(1, 6)	[[1. 1. 1. 0. 0. 0.]]
usuario	int32	(1, 6)	[[1 1 0 1 0 0]]

Note que o processo executado dentro do laço de repetição confere com a projeção anterior.

```
1
1
0
Matrix
1
0
0
```

Sendo assim, é impresso o nome do filme a ser recomendado a este usuário com base em suas preferências, nesse caso, Matrix.

Código Completo:

```
1 from rbm import RBM
2 import numpy as np
3
4 rbm = RBM(num_visible = 6,
5            num_hidden = 2)
6
7 base = np.array([[1,1,1,0,0,0],
8                  [1,0,1,0,0,0],
9                  [1,1,1,0,0,0],
10                 [0,0,1,1,1,1],
11                 [0,0,1,1,0,1],
12                 [0,0,1,0,1,0]])
13
14 filmes = ['O Exorcista',
15            'American Pie',
16            'Matrix',
17            'Forrest Gump',
18            'Documentário X',
19            'O Rei Leão']
20
21 rbm.train(base,
22            max_epochs = 3000)
23 rbm.weights
24
25 usuario = np.array([[1,1,0,1,0,0]])
26
27 rbm.run_visible(usuario)
28 camada_oculta = np.array([[0,1]])
29 recomendacao = rbm.run_hidden(camada_oculta)
30
31 for i in range(len(usuario[0])):
32     print(usuario[0,i])
33     if usuario[0,i] == 0 and recomendacao[0,i] == 1:
34         print(filmes[i])
```

35 – Parâmetros Adicionais

Ao longo de nossos exemplos abordados nesse livro, foram passados os modelos mais comumente usados, assim como sua estrutura e parametrização, porém, este é apenas o passo inicial, consultando a documentação das bibliotecas usadas você verá que existe literalmente uma infinidade de parâmetros/argumentos que podem ser utilizados além dos defaults e dos que programamos manualmente. Segue abaixo alguns exemplos, dependendo muito das aplicações de suas redes neurais, da forma como você terá que adaptá-las a alguma particularidade de algum problema, vale a pena se aprofundar neste quesito, descobrir novos parâmetros e realizar testes com os mesmos. Lembre-se que aqui nossa abordagem foi a mais simples, direta e prática possível dentro de uma das áreas de maior complexidade na computação.

`np.array()`

Arguments
<code>array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)</code>

```
pd.read_csv()
```

Arguments

```
parser_f(filepath_or_buffer, sep=sep,
         delimiter=None, # Column and Index
         Locations and Names header='infer',
         names=None, index_col=None, usecols=None,
         squeeze=False, prefix=None,
         mangle_dupe_cols=True, # General Parsing
         Configuration dtype=None, engine=None,
         converters=None, true_values=None,
         false_values=None,
         skipinitialspace=False, skiprows=None,
         skipfooter=0, nrows=None, # NA and
         Missing Data Handling na_values=None,
         keep_default_na=True, na_filter=True,
         verbose=False, skip_blank_lines=True, #
         Datetime Handling parse_dates=False,
         infer_datetime_format=False,
         keep_date_col=False, date_parser=None,
         dayfirst=False, # Iteration
         iterator=False, chunksize=None, #
         Quoting, Compression, and File Format
         compression='infer', thousands=None,
         decimal=b'.', lineterminator=None,
         quotechar="'", quoting=csv.QUOTE_MINIMAL,
         doublequote=True, escapechar=None,
         comment=None, encoding=None,
         dialect=None, tupleize_cols=None, # Error
         Handling error_bad_lines=True,
         warn_bad_lines=True, # Internal
         delim_whitespace=False, low_memory=_c_par-
         ser_defaults['low_memory'],
```

```
Sequential()
```

Arguments

```
Sequential(self, layers=None, name=None)
```

```
.add(Dense(1))
```

Arguments

```
Dense(self, units, activation=None, use_bias=True,
       kernel_initializer='glorot_uniform',
       bias_initializer='zeros',
       kernel_regularizer=None,
       bias_regularizer=None,
       activity_regularizer=None,
       kernel_constraint=None,
       bias_constraint=None, **kwargs)
```

```
.fit()
```

Arguments

```
fit(self, x=None, y=None, batch_size=None,  
    epochs=1, verbose=1, callbacks=None,  
    validation_split=0., validation_data=None,  
    shuffle=True, class_weight=None,  
    sample_weight=None, initial_epoch=0,  
    steps_per_epoch=None, validation_steps=None,  
    **kwargs)
```

```
.add(LSTM())
```

Arguments

```
LSTM(self, units, activation='tanh',  
      recurrent_activation='hard_sigmoid',  
      use_bias=True,  
      kernel_initializer='glorot_uniform',  
      recurrent_initializer='orthogonal',  
      bias_initializer='zeros',  
      unit_forget_bias=True,  
      kernel_regularizer=None,  
      recurrent_regularizer=None,  
      bias_regularizer=None,  
      activity_regularizer=None,  
      kernel_constraint=None,  
      recurrent_constraint=None,  
      bias_constraint=None, dropout=0.,  
      recurrent_dropout=0., implementation=1,  
      return_sequences=False, return_state=False,  
      go_backwards=False, stateful=False,  
      unroll=False, **kwargs)
```

```
.add(Dropout())  
    Arguments  
    Dropout(self, rate, noise_shape=None, seed=None,  
           **kwargs)  
ImageDataGenerator()  
    Arguments  
    ImageDataGenerator(self, featurewise_center=False,  
                      samplewise_center=False, featur  
                      ewise_std_normalization=False,  
                      samplewise_std_normalization=Fa  
                      lse, zca_whitening=False,  
                      zca_epsilon=1e-6,  
                      rotation_range=0.,  
                      width_shift_range=0.,  
                      height_shift_range=0.,  
                      brightness_range=None,  
                      shear_range=0., zoom_range=0.,  
                      channel_shift_range=0.,  
                      fill_mode='nearest', cval=0.,  
                      horizontal_flip=False,  
                      vertical_flip=False,  
                      rescale=None,  
                      preprocessing_function=None,  
                      data_format=None,  
                      validation_split=0.0,  
                      dtype=None)
```

Repare que em nossos exemplos contamos muito com estes parâmetros pré-configurados, que de modo geral funcionam muito bem, mas é interessante saber que sim existe toda uma documentação sobre cada mecanismo interno de cada ferramenta dessas, para que possamos evoluir nossos modelos adaptando-os a toda e qualquer particularidade.

36 – Capítulo Resumo

Muito bem, foram muitas páginas, muitos exemplos, muitas linhas de código, mas acredito que neste momento você já tenha construído bases bastante sólidas sobre o que é ciência de dados e aprendizado de máquina.

Como nossa metodologia foi procedural, gradativamente fomos abordando tópicos que nos levaram desde uma tabela verdade até uma rede neural artificial convolucional gerativa, algo que está enquadrado no que há de mais avançado dentro dessa área da computação.

Apenas agora dando uma breve resumida nos conceitos, desde o básico ao avançado, espero que você realmente não tenha dúvidas quanto a teoria e quanto a aplicação de tais recursos. Caso ainda haja, sempre é interessante consultar as documentações oficiais de cada biblioteca assim como buscar exemplos diferentes dos abordados neste livro. Se tratando de programação, conhecimento nunca será demais, sempre haverá algo novo a aprender ou ao menos aprimorar.

Pois bem, abaixo está a lista de cada ferramenta que usamos (por ordem de uso) com sua explicação resumida:

BIBLIOTECAS E MÓDULOS

Numpy: Biblioteca de computação científica que suporta operações matemáticas para dados em forma vetorial ou matricial (arrays).

SKLearn: Biblioteca que oferece uma série de ferramentas para leitura e análise de dados.

Pandas: Biblioteca desenvolvida para manipulação e análise de dados, em especial, importadas do formato .csv

Keras: Biblioteca que oferece uma série de modelos estruturados para fácil construção de redes neurais artificiais.

MatPlotLib: Biblioteca integrada ao Numpy para exibição de dados em forma de gráficos.

KNeighborsClassifier: Módulo pré-configurado para aplicação do modelo estatístico KNN.

TrainTestSplit: Módulo pré-configurado para divisão de amostras em grupos, de forma a testá-los individualmente e avaliar a performance do algoritmo.

Metrics: Módulo que oferece ferramentas para avaliação de performance dos processos aplicados sobre uma rede neural.

LogisticRegression: Módulo que aplica a chamada regressão logística, onde se categorizam os dados para que a partir dos mesmos se possam realizar previsões.

LabelEncoder: Módulo que permite a alteração dos rótulos dos objetos/variáveis internamente de forma que se possa fazer o correto cruzamento dos dados.

Sequential: Módulo que permite facilmente criar estruturas de redes neurais sequenciais, onde os nós das camadas anteriores são ligados com os da camada subsequente.

Dense: Módulo que permite que se criem redes neurais densas, modelo onde todos neurônios estão (ou podem estar) interligados entre si.

KerasClassifier: Módulo que oferece um modelo de classificador de amostras pré-configurado e de alta eficiência a partir da biblioteca Keras.

NPUtils: Módulo que oferece uma série de ferramentas adicionais para se trabalhar e extrair maiores informações a partir de arrays.

Dropout: Módulo que funciona como filtro para redes que possuem muitos neurônios por camada em sua estrutura, ele aleatoriamente remove neurônios para que sejam testadas a integridade dos dados dos neurônios remanescentes.

GridSearchCV: Módulo que aplica técnicas de validação cruzada sobre matrizes, podendo fazer o uso de vários parâmetros e comparar qual o que resulta em dados mais íntegros.

OneHotEncoder: Módulo que permite a criação de variáveis do tipo dummy, variáveis estas com indexação interna própria e para fácil normalização dos tipos de dados.

KerasRegressor: Módulo que oferece um modelo pré-configurado de regressor a partir da biblioteca Keras.

Conv2D: Módulo que permite a transformação de arrays tridimensionais para estruturas bidimensionais para se aplicar certas funções limitadas a 2D.

MaxPooling2D: Módulo que oferece ferramentas para que sejam filtrados dados fora de um padrão, a fim de aumentar a precisão dos resultados.

Flatten: Módulo que permite a redução de dados matriciais de duas ou mais dimensões para uma dimensão, muito usado para vetorializar dados de uma imagem transformando-os em neurônios de uma determinada camada.

BatchNormalization: Módulo que possui internamente ferramentas para que se encontre os melhores padrões de atualização de pesos e valores dos registros de uma rede neural.

ImageDataGenerator: Módulo que permite a conversão e extração de atributos numéricos a partir de imagens para geração de novas imagens.

MinMaxScaler: Módulo muito usado quando necessária a normalização do range de dados dentro de um intervalo pré-estabelecido (entre 0 e 1, por exemplo).

LSTM: Biblioteca que permite a criação de modelos de redes neurais recorrentes. Onde a chamada recorrência se dá pela atualização dos valores do próprio neurônio na própria camada após aplicadas algumas funções.

MiniSom: Biblioteca que possui ferramentas para fácil criação de mapas auto organizáveis.

PColor: Módulo que permite a inserção de marcadores coloridos em cima de gráficos.

ColorBar: Módulo acessório ao PColor que permite a inserção de uma barra de cores que gera um padrão mínimo/máximo de acordo com as características e proximidade dos dados.

RBM: Módulo que permite a criação de modelos de Boltzmann Machines, ou seja, de redes que funcionam como sistemas de recomendação.

37 – Considerações Finais

E assim concluímos esse pequeno tratado introdutório ao processo de aprendizado de máquina via redes neurais artificiais. Foram mais de 200 páginas de conteúdo puro, mais de 500 linhas de código com mais de 450 ilustrações passo-a-passo, e o que posso dizer a este momento é que de fato foi muito gratificante de minha parte ter escrito esse pequeno compêndio de ideias, que este livro realmente tenha lhe ajudado a dar seus primeiros passos dentro dessa incrível área da computação. Por fim, resta meu agradecimento por ter comprado esta humilde obra e que a leitura da mesma tenha sido tão prazerosa a você quanto foi a mim por escrevê-la.

Esse é apenas o início de uma grande jornada dentro da computação científica, lhe desejo sucesso e que sua vida profissional neste meio seja de grandes realizações.

Qualquer crítica, dúvida ou sugestão sinta-se à vontade para entrar em contato comigo através do e-mail: fernando2rad@gmail.com

Um forte abraço,

Fernando Feltrin