

***INTELIGÊNCIA ARTIFICIAL***

**UA02 / LABORATÓRIO # 1**

**ASSUNTO:** PREPARANDO AMBIENTE PYTHON+VSCODE / USO DO PANDAS

**Materiais de Apoio**

**Site oficial do Scikit Learn:**

**scikit-learn.org/**

**Site oficial do Projeto Jupyter:**

**http://jupyter.org/**

**Vídeo BEM introdutório sobre Jupyter:**

**https://www.youtube.com/watch?v=m0FbNlhNyQ8**

**Parte 1 – Preparação do Ambiente Jupyter + VS Code e iniciando os Trabalhos**

***Dataset usado:*** *nenhum*

1. Instalar Python versão 3.10.
2. Na linha de comando do Sistema Operacional (que pode ser acessível também pelo VS Code), instalar as bibliotecas necessárias para o trabalho com Data Science em nossa disciplina de Inteligência Artificial:

*pip install numpy*

*pip install scipy*

*pip install pandaspip*

*pip install matplotlib*

*pip install scikit-learn*

*pip install seaborn*

*pip install tensorflow*

**Obs.** em caso de não funcionar “pip install”, tentar “py –m pip install”

python -m pip install --upgrade pip

1. Instalar Microsoft VS Code na sua máquina’.

<https://code.visualstudio.com/download>

1. Acessar o VS Code para instalar extensões (VS Code):

- **Jupyter (Microsoft)**: “Jupyter notebook support, interactive programming and computing that supports Intellisense, debugging and more.”

- **Python (Microsoft)**: “IntelliSense (Pylance), Linting, Debugging (multi-threaded, remote), Jupyter Notebooks, code formatting, refactoring, unit tests, and more”

- **Jupyter Notebook Renderers (Microsoft):** “Renderers for Jupyter Notebooks (with plotly, vega, gif, png, svg, jpeg and other such outputs)”

1. No VS Code, vamos criar um “Novo Arquivo..” do tipo “Jupyter Notebook”:

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

1. Testar o ambiente com o seguinte código:

*import seaborn as sns*

*import matplotlib as plt*

*df = sns.load\_dataset("penguins")*

*sns.jointplot(data=df, x="flipper\_length\_mm", y="bill\_length\_mm", hue="species")*

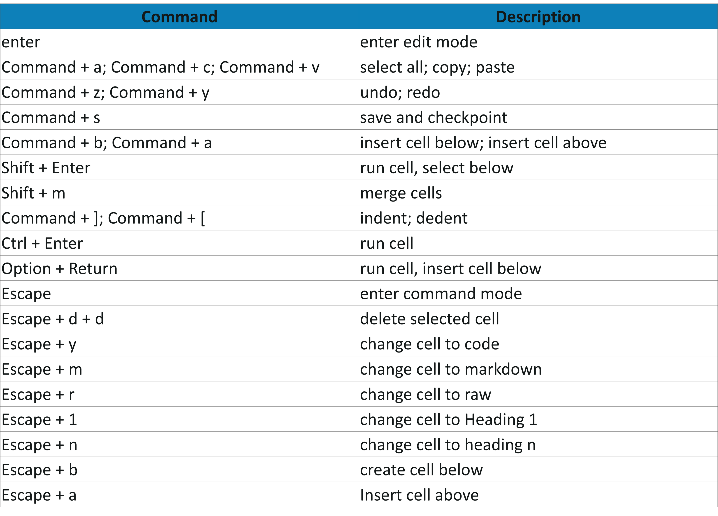
**Obs**. em caso de o VS Code não reconhecer o interpretador Python, acessar:

File > Preferences > Settings > Extensions

Depois selecionar “Python” e no campo “Default Interpreter Path” inserir seu path:

“C:\Users\Walmir\AppData\Local\Programs\Python\Python310\python.exe”

Atalhos úteis do Jupyter Notebook:



**Dica:** Pode-se usar o formato *Markdown* para documentar o Projeto!!

**Ambiente Virtual Python (venv)**:

**Para criar:**

python -m venv nome\_do\_ambiente

*Exemplo: python -m venv ambml1*

Será criada uma pasta com todas as dependências padrão do *Python*.

**Para ativar:**

*nome\_da\_virtualenv\Scripts\activate*

*Exemplo: C:\IA-Estudos\PLN\ambml1\Scripts\activate*

*Obs. Poderemos então, fazer instalações que ficarão internamente nesta “venv”, um ambiente independente, sem interferir em outras “venv” e na instalação Python da máquina.*

**Parte 2 – Pandas – Introdução ao uso da Biblioteca – Tutorial do Kaggle**

<https://www.kaggle.com/learn/pandas>

**1. CREATING, READING AND WRITING**

Introduction

In this micro-course, you'll learn all about **[pandas](https://pandas.pydata.org/)**, the most popular Python library for data analysis.

Along the way, you'll complete several hands-on exercises with real-world data. We recommend that you work on the exercises while reading the corresponding tutorials.

**To start the first exercise, please click [here](https://www.kaggle.com/kernels/fork/587970" \t "_top).**

In this tutorial, you will learn how to create your own data, along with how to work with data that already exists.

Getting started

To use pandas, you'll typically start with the following line of code.

In [1]:

import pandas as pd

Creating data

There are two core objects in pandas: the **DataFrame** and the **Series**.

DataFrame

A DataFrame is a table. It contains an array of individual *entries*, each of which has a certain *value*. Each entry corresponds to a row (or *record*) and a *column*.

For example, consider the following simple DataFrame:

In [2]:

pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})

Out[2]:

|  | Yes | No |
| --- | --- | --- |
| 0 | 50 | 131 |
| 1 | 21 | 2 |

In this example, the "0, No" entry has the value of 131. The "0, Yes" entry has a value of 50, and so on.

DataFrame entries are not limited to integers. For instance, here's a DataFrame whose values are strings:

In [3]:

pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'], 'Sue': ['Pretty good.', 'Bland.']})

Out[3]:

|  | Bob | Sue |
| --- | --- | --- |
| 0 | I liked it. | Pretty good. |
| 1 | It was awful. | Bland. |

We are using the pd.DataFrame() constructor to generate these DataFrame objects. The syntax for declaring a new one is a dictionary whose keys are the column names (Bob and Sue in this example), and whose values are a list of entries. This is the standard way of constructing a new DataFrame, and the one you are most likely to encounter.

The dictionary-list constructor assigns values to the *column labels*, but just uses an ascending count from 0 (0, 1, 2, 3, ...) for the *row labels*. Sometimes this is OK, but oftentimes we will want to assign these labels ourselves.

The list of row labels used in a DataFrame is known as an **Index**. We can assign values to it by using an index parameter in our constructor:

In [4]:

pd.DataFrame({'Bob': ['I liked it.', 'It was awful.'],

'Sue': ['Pretty good.', 'Bland.']},

index=['Product A', 'Product B'])

Out[4]:

|  | Bob | Sue |
| --- | --- | --- |
| Product A | I liked it. | Pretty good. |
| Product B | It was awful. | Bland. |

Series

A Series, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. And in fact you can create one with nothing more than a list:

In [5]:

pd.Series([1, 2, 3, 4, 5])

Out[5]:

0 1

1 2

2 3

3 4

4 5

dtype: int64

A Series is, in essence, a single column of a DataFrame. So you can assign column values to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name:

In [6]:

pd.Series([30, 35, 40], index=['2015 Sales', '2016 Sales', '2017 Sales'], name='Product A')

Out[6]:

2015 Sales 30

2016 Sales 35

2017 Sales 40

Name: Product A, dtype: int64

The Series and the DataFrame are intimately related. It's helpful to think of a DataFrame as actually being just a bunch of Series "glued together". We'll see more of this in the next section of this tutorial.

Reading data files

Being able to create a DataFrame or Series by hand is handy. But, most of the time, we won't actually be creating our own data by hand. Instead, we'll be working with data that already exists.

Data can be stored in any of a number of different forms and formats. By far the most basic of these is the humble CSV file. When you open a CSV file you get something that looks like this:

Product A,Product B,Product C,

30,21,9,

35,34,1,

41,11,11

So a CSV file is a table of values separated by commas. Hence the name: "Comma-Separated Values", or CSV.

Let's now set aside our toy datasets and see what a real dataset looks like when we read it into a DataFrame. We'll use the pd.read\_csv() function to read the data into a DataFrame. This goes thusly:

In [7]:

wine\_reviews = pd.read\_csv("../input/wine-reviews/winemag-data-130k-v2.csv")

We can use the shape attribute to check how large the resulting DataFrame is:

In [8]:

wine\_reviews.shape

Out[8]:

(129971, 14)

So our new DataFrame has 130,000 records split across 14 different columns. That's almost 2 million entries!

We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows:

In [9]:

wine\_reviews.head()

Out[9]:

|  | Unnamed: 0 | country | description | designation | points | price | province | region\_1 | region\_2 | taster\_name | taster\_twitter\_handle | title | variety | winery |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | Italy | Aromas include tropical fruit, broom, brimston... | Vulkà Bianco | 87 | NaN | Sicily & Sardinia | Etna | NaN | Kerin O’Keefe | @kerinokeefe | Nicosia 2013 Vulkà Bianco (Etna) | White Blend | Nicosia |
| 1 | 1 | Portugal | This is ripe and fruity, a wine that is smooth... | Avidagos | 87 | 15.0 | Douro | NaN | NaN | Roger Voss | @vossroger | Quinta dos Avidagos 2011 Avidagos Red (Douro) | Portuguese Red | Quinta dos Avidagos |
| 2 | 2 | US | Tart and snappy, the flavors of lime flesh and... | NaN | 87 | 14.0 | Oregon | Willamette Valley | Willamette Valley | Paul Gregutt | @paulgwine | Rainstorm 2013 Pinot Gris (Willamette Valley) | Pinot Gris | Rainstorm |
| 3 | 3 | US | Pineapple rind, lemon pith and orange blossom ... | Reserve Late Harvest | 87 | 13.0 | Michigan | Lake Michigan Shore | NaN | Alexander Peartree | NaN | St. Julian 2013 Reserve Late Harvest Riesling ... | Riesling | St. Julian |
| 4 | 4 | US | Much like the regular bottling from 2012, this... | Vintner's Reserve Wild Child Block | 87 | 65.0 | Oregon | Willamette Valley | Willamette Valley | Paul Gregutt | @paulgwine | Sweet Cheeks 2012 Vintner's Reserve Wild Child... | Pinot Noir | Sweet Cheeks |

The pd.read\_csv() function is well-endowed, with over 30 optional parameters you can specify. For example, you can see in this dataset that the CSV file has a built-in index, which pandas did not pick up on automatically. To make pandas use that column for the index (instead of creating a new one from scratch), we can specify an index\_col.

In [10]:

wine\_reviews = pd.read\_csv("../input/wine-reviews/winemag-data-130k-v2.csv", index\_col=0)

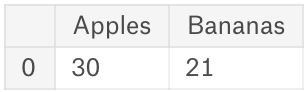
wine\_reviews.head()

Out[10]:

|  | country | description | designation | points | price | province | region\_1 | region\_2 | taster\_name | taster\_twitter\_handle | title | variety | winery |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Italy | Aromas include tropical fruit, broom, brimston... | Vulkà Bianco | 87 | NaN | Sicily & Sardinia | Etna | NaN | Kerin O’Keefe | @kerinokeefe | Nicosia 2013 Vulkà Bianco (Etna) | White Blend | Nicosia |
| 1 | Portugal | This is ripe and fruity, a wine that is smooth... | Avidagos | 87 | 15.0 | Douro | NaN | NaN | Roger Voss | @vossroger | Quinta dos Avidagos 2011 Avidagos Red (Douro) | Portuguese Red | Quinta dos Avidagos |
| 2 | US | Tart and snappy, the flavors of lime flesh and... | NaN | 87 | 14.0 | Oregon | Willamette Valley | Willamette Valley | Paul Gregutt | @paulgwine | Rainstorm 2013 Pinot Gris (Willamette Valley) | Pinot Gris | Rainstorm |
| 3 | US | Pineapple rind, lemon pith and orange blossom ... | Reserve Late Harvest | 87 | 13.0 | Michigan | Lake Michigan Shore | NaN | Alexander Peartree | NaN | St. Julian 2013 Reserve Late Harvest Riesling ... | Riesling | St. Julian |
| 4 | US | Much like the regular bottling from 2012, this... | Vintner's Reserve Wild Child Block | 87 | 65.0 | Oregon | Willamette Valley | Willamette Valley | Paul Gregutt | @paulgwine | Sweet Cheeks 2012 Vintner's Reserve Wild Child... | Pinot Noir | Sweet Cheeks |

Exercises

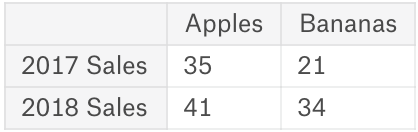
## **1.** In the cell below, create a DataFrame fruits that looks like this:



# Your code goes here. Create a dataframe matching the above diagram and assign it to the variable fruits.

fruits = pd.DataFrame({'Apples':[30], 'Bananas':[21]})

## **2.** Create a dataframe fruit\_sales that matches the diagram below:



# Your code goes here. Create a dataframe matching the above diagram and assign it to the variable fruit\_sales.

fruit\_sales = pd.DataFrame({'Apples':[35,41], 'Bananas':[21,34]}, index=['2017 Sales','2018 Sales'])

## **3.** Create a variable ingredients with a Series that looks like:

Flour 4 cups

Milk 1 cup

Eggs 2 large

Spam 1 can

Name: Dinner, dtype: object

ingredients = pd.Series(['4 cups','1 cup','2 large','1 can',],index=['Flour','Milk','Eggs','Spam'], name='Dinner')

## **4.** Read the following csv dataset of wine reviews into a DataFrame called reviews:



The filepath to the csv file is ../input/wine-reviews/winemag-data\_first150k.csv. The first few lines look like:

reviews = pd.read\_csv("../input/wine-reviews/winemag-data-130k-v2.csv")

## **5.** Run the cell below to create and display a DataFrame called animals:

animals **=** pd.DataFrame({'Cows': [12, 20], 'Goats': [22, 19]}, index**=**['Year 1', 'Year 2'])

| **Cows** | **Goats** |
| --- | --- |
| **Year 1** | 12 | 22 |
| **Year 2** | 20 | 19 |

In the cell below, write code to save this DataFrame to disk as a csv file with the name cows\_and\_goats.csv.

# Your code goes here

animals.to\_csv("cows\_and\_goats.csv")

**2. INDEXING, SELECTING AND ASSIGNING**

The pd.read\_csv() function is well-endowed, with over 30 optional parameters you can

Introduction

Selecting specific values of a pandas DataFrame or Series to work on is an implicit step in almost any data operation you'll run, so one of the first things you need to learn in working with data in Python is how to go about selecting the data points relevant to you quickly and effectively.

Naive accessors

Native Python objects provide good ways of indexing data. Pandas carries all of these over, which helps make it easy to start with.

Consider this DataFrame:

In [2]:

reviews

Out[2]:

|  | country | description | designation | points | price | province | region\_1 | region\_2 | taster\_name | taster\_twitter\_handle | title | variety | winery |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Italy | Aromas include tropical fruit, broom, brimston... | Vulkà Bianco | 87 | NaN | Sicily & Sardinia | Etna | NaN | Kerin O’Keefe | @kerinokeefe | Nicosia 2013 Vulkà Bianco (Etna) | White Blend | Nicosia |
| 1 | Portugal | This is ripe and fruity, a wine that is smooth... | Avidagos | 87 | 15.0 | Douro | NaN | NaN | Roger Voss | @vossroger | Quinta dos Avidagos 2011 Avidagos Red (Douro) | Portuguese Red | Quinta dos Avidagos |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 129969 | France | A dry style of Pinot Gris, this is crisp with ... | NaN | 90 | 32.0 | Alsace | Alsace | NaN | Roger Voss | @vossroger | Domaine Marcel Deiss 2012 Pinot Gris (Alsace) | Pinot Gris | Domaine Marcel Deiss |
| 129970 | France | Big, rich and off-dry, this is powered by inte... | Lieu-dit Harth Cuvée Caroline | 90 | 21.0 | Alsace | Alsace | NaN | Roger Voss | @vossroger | Domaine Schoffit 2012 Lieu-dit Harth Cuvée Car... | Gewürztraminer | Domaine Schoffit |

129971 rows × 13 columns

In Python, we can access the property of an object by accessing it as an attribute. A book object, for example, might have a title property, which we can access by calling book.title. Columns in a pandas DataFrame work in much the same way.

Hence to access the country property of reviews we can use:

In [3]:

reviews.country

Out[3]:

0 Italy

1 Portugal

...

129969 France

129970 France

Name: country, Length: 129971, dtype: object

If we have a Python dictionary, we can access its values using the indexing ([]) operator. We can do the same with columns in a DataFrame:

In [4]:

reviews['country']

0 Italy

1 Portugal

...

129969 France

129970 France

Name: country, Length: 129971, dtype: object

hese are the two ways of selecting a specific Series out of a DataFrame. Neither of them is more or less syntactically valid than the other, but the indexing operator [] does have the advantage that it can handle column names with reserved characters in them (e.g. if we had a country providence column, reviews.country providence wouldn't work).

Doesn't a pandas Series look kind of like a fancy dictionary? It pretty much is, so it's no surprise that, to drill down to a single specific value, we need only use the indexing operator [] once more:

In [5]:

reviews['country'][0]

Out[5]:

'Italy'

Indexing in pandas

The indexing operator and attribute selection are nice because they work just like they do in the rest of the Python ecosystem. As a novice, this makes them easy to pick up and use. However, pandas has its own accessor operators, loc and iloc. For more advanced operations, these are the ones you're supposed to be using.

### Index-based selection

Pandas indexing works in one of two paradigms. The first is **index-based selection**: selecting data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we may use the following:

In [6]:

reviews.iloc[0]

Out[6]:

country Italy

description Aromas include tropical fruit, broom, brimston...

...

variety White Blend

winery Nicosia

Name: 0, Length: 13, dtype: object

Both loc and iloc are row-first, column-second. This is the opposite of what we do in native Python, which is column-first, row-second.

This means that it's marginally easier to retrieve rows, and marginally harder to get retrieve columns. To get a column with iloc, we can do the following:

In [7]:

reviews.iloc[:, 0]

Out[7]:

0 Italy

1 Portugal

...

129969 France

129970 France

Name: country, Length: 129971, dtype: object

On its own, the : operator, which also comes from native Python, means "everything". When combined with other selectors, however, it can be used to indicate a range of values. For example, to select the country column from just the first, second, and third row, we would do:

reviews.iloc[:3, 0]

Out[8]:

0 Italy

1 Portugal

2 US

Name: country, dtype: object

Or, to select just the second and third entries, we would do:

In [9]:

reviews.iloc[1:3, 0]

Out[9]:

1 Portugal

2 US

Name: country, dtype: object

It's also possible to pass a list:

In [10]:

reviews.iloc[[0, 1, 2], 0]

Out[10]:

0 Italy

1 Portugal

2 US

Name: country, dtype: object

Finally, it's worth knowing that negative numbers can be used in selection. This will start counting forwards from the *end* of the values. So for example here are the last five elements of the dataset.

reviews.iloc[-5:]

Out[11]:

|  | country | description | designation | points | price | province | region\_1 | region\_2 | taster\_name | taster\_twitter\_handle | title | variety | winery |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 129966 | Germany | Notes of honeysuckle and cantaloupe sweeten th... | Brauneberger Juffer-Sonnenuhr Spätlese | 90 | 28.0 | Mosel | NaN | NaN | Anna Lee C. Iijima | NaN | Dr. H. Thanisch (Erben Müller-Burggraef) 2013 ... | Riesling | Dr. H. Thanisch (Erben Müller-Burggraef) |
| 129967 | US | Citation is given as much as a decade of bottl... | NaN | 90 | 75.0 | Oregon | Oregon | Oregon Other | Paul Gregutt | @paulgwine | Citation 2004 Pinot Noir (Oregon) | Pinot Noir | Citation |
| 129968 | France | Well-drained gravel soil gives this wine its c... | Kritt | 90 | 30.0 | Alsace | Alsace | NaN | Roger Voss | @vossroger | Domaine Gresser 2013 Kritt Gewurztraminer (Als... | Gewürztraminer | Domaine Gresser |
| 129969 | France | A dry style of Pinot Gris, this is crisp with ... | NaN | 90 | 32.0 | Alsace | Alsace | NaN | Roger Voss | @vossroger | Domaine Marcel Deiss 2012 Pinot Gris (Alsace) | Pinot Gris | Domaine Marcel Deiss |
| 129970 | France | Big, rich and off-dry, this is powered by inte... | Lieu-dit Harth Cuvée Caroline | 90 | 21.0 | Alsace | Alsace | NaN | Roger Voss | @vossroger | Domaine Schoffit 2012 Lieu-dit Harth Cuvée Car... | Gewürztraminer | Domaine Schoffit |

### Label-based selection

The second paradigm for attribute selection is the one followed by the loc operator: **label-based selection**. In this paradigm, it's the data index value, not its position, which matters.

For example, to get the first entry in reviews, we would now do the following:

In [12]:

reviews.loc[0, 'country']

Out[12]:

'Italy'

iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use iloc we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. loc, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using loc instead. For example, here's one operation that's much easier using loc:

In [13]:

reviews.loc[:, ['taster\_name', 'taster\_twitter\_handle', 'points']]

Out[13]:

|  | taster\_name | taster\_twitter\_handle | points |
| --- | --- | --- | --- |
| 0 | Kerin O’Keefe | @kerinokeefe | 87 |
| 1 | Roger Voss | @vossroger | 87 |
| ... | ... | ... | ... |
| 129969 | Roger Voss | @vossroger | 90 |
| 129970 | Roger Voss | @vossroger | 90 |

129971 rows × 3 columns

### Choosing between loc and iloc

When choosing or transitioning between loc and iloc, there is one "gotcha" worth keeping in mind, which is that the two methods use slightly different indexing schemes.

iloc uses the Python stdlib indexing scheme, where the first element of the range is included and the last one excluded. So 0:10 will select entries 0,...,9. loc, meanwhile, indexes inclusively. So 0:10 will select entries 0,...,10.

Why the change? Remember that loc can index any stdlib type: strings, for example. If we have a DataFrame with index values Apples, ..., Potatoes, ..., and we want to select "all the alphabetical fruit choices between Apples and Potatoes", then it's a lot more convenient to index df.loc['Apples':'Potatoes'] than it is to index something like df.loc['Apples', 'Potatoet] (t coming after s in the alphabet).

This is particularly confusing when the DataFrame index is a simple numerical list, e.g. 0,...,1000. In this case df.iloc[0:1000] will return 1000 entries, while df.loc[0:1000] return 1001 of them! To get 1000 elements using loc, you will need to go one lower and ask for df.loc[0:999].

Otherwise, the semantics of using loc are the same as those for iloc.

**Bom Trabalho!!**