## Análise e Visualização de Dados com Python

Nesta aula você vai aprender a empregar o Python para fazer boa parte das operações que você já aprendeu em R. Aqui você aprenderá:

* Como selecionar dados tabulares com Python
* Obter as principais estatísticas descritivas dos dados
* Produzir gráficos de tendência, de distribuição, relações e proporções

# Introdução

R e Python são as linguagens mais empregadas para Análise Exploratória de Dados e Ciência de Dados e a opção de uma ou outra depende muito mais, em geral, do contexto de uso que ser quer fazer. Por isso, aqui, aprenderemos a empregar as duas linguagens.

*Embora pareça haver uma predominância do Python nos últimos anos isso pode depender da área de domínio em que estamos interessados. O R ainda parece dominar as aplicações em áreas como bioinformática e ciências sociais, onde há muito tempo são aplicados modelos estatísticos com R. Já aplicações de comércio eletrônico e marketing, assim como modelos que tratam imagens ou linguagem natural, o Python em geral parece ser uma melhor solução.*

*Por que essas linguagens e não linguagens como Java ou C? A Análise de Dados e a Ciência de Dados tem algumas características que as diferem do desenvolvimento tradicional de aplicações. A integração com recursos para Web ou para aplicações Mobile, por exemplo, é essencial para o desenvolvimento aplicações modernas, mas tem um papel bastante secundário na Análise de Dados. Além disso, aplicações tradicionais são voltadas em muitos casos para o processamento transacional, empregando quantidades relativamente pequenas de dados. Por outro lado, script languages e tipos de dados fracos, como R e Python, parecem ser mais vantajosos quando queremos experimentar diferentes execuções para exploração de dados que não temos clareza ainda de tipos ou valores empregados.*

*Existem também muitas ferramentas para Análise e Ciência de Dados (soluções de spreadsheet view of data, drag-and-drop tools com funções pré-definidas, dentre outras) e talvez você esteja familiarizado com alguns nomes como SAS, PowerBI, Tableau, Qlik, SPSS etc. e, não obstante o uso bastante comum dessas soluções elas são bastante proprietárias e o uso de linguagens de programação oferece, em geral, mais flexibilidade, independência e possibilidade de reuso. De qualquer modo essas ferramentas permitem proporcionar muitos primeiros resultados de forma muito rápida e não devem ser descartadas se estiverem disponíveis.*

*Há também outras linguagens como Scala, Julia e o próprio Java que apresentam uso em Ciência de Dados em alguns contextos mais específicos e, ao final, o uso de uma solução ou outra deve ser analisada caso a caso sem, necessariamente, assumirmos uma ou outra como melhor e, na maior parte dos casos empregaremos muitas soluções conjuntas.*

Neste ponto você já tem conhecimento dos principais conceitos de Python e de estruturas de dados como listas e dicionários e partiremos, então, desse ponto. Também partiremos do princípio que você já sabe da importância de se fazer seleções dos dados e das técnicas de visualização que aprendemos nas aulas anteriores com R e poderemos nos aprofundar um pouco mais em alguns temas.

Assi, nesta aula, você irá aprender como empregar o pacote Pandas do Python para adquirir dados, selecionar dados de interesse e fazer algumas transformações simples, mas úteis sobre os dados. Transformações mais elaboradas, como merge e reshape de dados você deverá ver mais adiante no curso. Você também aprenderá a empregar as bibliotecas Matplotlib e Seaborn para criar visualizações dos dados para compreender grandes conjuntos de dados, à exemplo do que vimos em R.

# Pandas

O Pandas é um projeto de código aberto desenvolvido inicialmente por Wes McKinney a partir de 2008 que empregou como base para seu desenvolvimento as estruturas de dataframes já presentes na linguagem R.

O Pandas é uma biblioteca para manipulação e análise de dados orientada a colunas. Ele organiza os dados em formato de tabelas, semelhante ao que você encontra em bancos de dados relacionais ou mesmo em sistemas de planilhas. Sobre essas estruturas e ele fornece uma série de operações que tornam ele o principal pacote para manipulação de dados em Python. Outros pacotes de manipulação de dados, como o Dask e o PySpark, incorporam outros recursos como paralelismo para lidar com grandes conjuntos de dados mas guardam grande semelhança com o Pandas em termos de programação. Além disso o Pandas fornece acesso a inúmeros formatos de dados, de planilhas a bancos de dados NoSQL, e desse modo, conhecendo o Pandas você será capaz de manipular quaisquer dados em Python.

O Pandas foi desenvolvido sobre NumPy, para manipulação e análise de dados orientada a colunas. Ele organiza os dados em dois tipos de estruturas, Séries e DataFrames e opera seleções e transformações sobre essas coleções de dados.

* **DataFrame**: você pode imaginar como uma tabela de dados relacionais, com linhas e colunas nomeadas, como no Excel ou em SQL.
* **Series**: é uma única coluna de dados, e o DataFrame contém um ou mais Series com um nome para cada uma delas.

O DataFrame, implementado em Python, é uma abstração comumente usada para manipulação de dados que tem origem na linguagem R, mas você encontrará implementações similares também em outras linguagens.

## Import da biblioteca

import pandas as pd  
import numpy as np

## Criando um pd.Series e um pd.DataFrame

Uma pd.Series pode ser criada a partir de uma lista e um pd.Dataframe pode ser criado empregando-se uma estrutura de dicionário.

# Séries  
student\_names = pd.Series(['Adriana', 'Carol', 'Daniel'])  
age = pd.Series([18, 19, 19])  
  
# DataFrame  
students = pd.DataFrame({ 'Student Name': student\_names, 'Age': age })  
students

Student Name Age  
0 Adriana 18  
1 Carol 19  
2 Daniel 19

## Lendo um DataFrame

O uso mais comum do Pandas que faremos é para o acesso a arquivos e bases de dados e o Pandas permite ler e gravar dados de diferentes fontes como arquivos .csv, .json, .xlsx e arquivos de bancos de dados sql, Mongo etc.

|  |  |  |
| --- | --- | --- |
| **Format** | **Read** | **Save** |
| csv | pd.read\_csv() | df.to\_csv() |
| json | pd.read\_json() | df.to\_json() |
| xlsx | pd.read\_excel() | df.to\_excel() |
| hdf | pd.read\_hdf() | df.to\_hdf() |
| sql | pd.read\_sql() | df.to\_sql() |

que podem ser acessados local ou pela internet.

tips = pd.read\_csv('https://raw.githubusercontent.com/Rogerio-mack/Visualizacao-de-Dados-em-Python/main/data/tips.csv')  
tips.head()

total\_bill tip sex smoker day time size  
0 16.99 1.01 Female No Sun Dinner 2  
1 10.34 1.66 Male No Sun Dinner 3  
2 21.01 3.50 Male No Sun Dinner 3  
3 23.68 3.31 Male No Sun Dinner 2  
4 24.59 3.61 Female No Sun Dinner 4

O comando df.head() permite exibir o aspecto inicial dos dados a partir de suas primeiras linhas.

## Upload de arquivos de seu sistema de arquivos local

Se você estiver empregando o Google Colab e quiser acessar arquivos locais da sua máquina você pode *arrastar* os arquivos para o diretório da máquina virtual ou empregar uma API para acesso aos arquivos locais da sua máquina. files.upload retorna um dicionário dos arquivos que foram carregados. O dicionário é codificado pelo nome do arquivo e os valores são os dados que foram carregados.

Essas instruções encontram-se comentadas abaixo.

# from google.colab import files  
  
# uploaded = files.upload()  
  
# for fn in uploaded.keys():  
# print('User uploaded file "{name}" with length {length} bytes'.format(  
# name=fn, length=len(uploaded[fn])))

## Explorando a estrutura básica dos dados

Antes de explorarmos o conteúdo dos dados é útil termos algumas informações sobre a estrutura dos dados que estamos lidando como os valores que assumem os dados, quantas linhas temos em uma tabela, o número de atributos, o tipo de dados etc.

tips.shape # nr de linhas e colunas

(244, 7)

len(tips) # número de linhas

244

tips.columns.to\_list() # ou tips.columns

['total\_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size']

tips.dtypes # tipo dos atributos

total\_bill float64  
tip float64  
sex object  
smoker object  
day object  
time object  
size int64  
dtype: object

O comando describe() exibe informações estatísticas sumarizadas dos dados.

tips.describe(include='all')

total\_bill tip sex smoker day time size  
count 244.000000 244.000000 244 244 244 244 244.000000  
unique NaN NaN 2 2 4 2 NaN  
top NaN NaN Male No Sat Dinner NaN  
freq NaN NaN 157 151 87 176 NaN  
mean 19.785943 2.998279 NaN NaN NaN NaN 2.569672  
std 8.902412 1.383638 NaN NaN NaN NaN 0.951100  
min 3.070000 1.000000 NaN NaN NaN NaN 1.000000  
25% 13.347500 2.000000 NaN NaN NaN NaN 2.000000  
50% 17.795000 2.900000 NaN NaN NaN NaN 2.000000  
75% 24.127500 3.562500 NaN NaN NaN NaN 3.000000  
max 50.810000 10.000000 NaN NaN NaN NaN 6.000000

# Selecionando Dados

Seleções dos dados são bastante importantes. Você nem sempre estará interessado em exibir ou analisar todos os dados. Por exemplo, você pode ter dados de produção de várias unidades de uma fábrica, mas estar interessado somente em dados das unidades de São Paulo (seleção de linhas ou casos). Ou, você pode ter dados de vendas com diversas informações dos produtos (cor, modelo etc.) e dos clientes (nome, CPF etc.) e querer apenas dados de peso e dimensões do produto, e da origem e destino da compra para analisar os preços de frete (seleção de colunas ou atributos). Mais frequentemente ainda você vai realizar as duas seleções criando *slices* dos dados.

## Seleção de Colunas, Seleção de Linhas e de Linhas e Colunas

## Selecionando colunas de dados como pd.Series

Como vimos, as colunas de um DataFrame são séries do Pandas e podemos selecionar e operar com cada uma das séries de um DataFrame. Existem dois modos de seleção

df.<nome do atributo>  
  
df['<nome do atributo>']

eles retornam o mesmo resultado e a segunda forma é sempre necessária quando temos nomes de atributos com espaços ou caracteres especiais.

print(tips.tip)  
# ou  
# print(tips['tip'])

0 1.01  
1 1.66  
2 3.50  
3 3.31  
4 3.61  
 ...   
239 5.92  
240 2.00  
241 2.00  
242 1.75  
243 3.00  
Name: tip, Length: 244, dtype: float64

Esse conjunto de dados é um pd.Series e podemos fazer uma série de operações sobre essa coleção de dados, como calcular a soma ou a média de valores.

tip\_series = tips['tip']  
print(f'Média das gorjetas: USD {tip\_series.mean() :.2f} e Total das gorjetas: USD {tip\_series.sum() :.2f}')

Média das gorjetas: USD 3.00 e Total das gorjetas: USD 731.58

Pode ser útil também você transformar uma pd.Series em uma lista Python para que você possa realizar as operações de listas que você já conhece.

dias = tips.day  
dias = dias.to\_list()  
dias.count('Sat')

87

## Selecionando uma ou Mais Colunas como pd.DataFrame



Mas você pode estar interessado na seleção de *mais que uma coluna* de dados. Para isso você pode simplesmente informar para o Pandas uma **lista de atributos** a serem selecionados.

df [ [ <lista-de-colunas> ] ]

Diferentemente da seleção anterior que retorna um Série do Pandas, a seleção desse modo, com uma ou mais colunas informadas em uma lista, retorna um DataFrame.

A seleção como DataFrame permite a seleção de várias séries de dados (atributos) para construírmos subconjuntos de dados de interesse. Essas seleções de dados também são DataFrame e todas as operações de DataFrame são aplicáveis.

tips.columns

Index(['total\_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'], dtype='object')

selected\_tips = tips[['total\_bill', 'tip', 'size']]  
  
selected\_tips.head() # um novo dataframe com atributos selecionados

total\_bill tip size  
0 16.99 1.01 2  
1 10.34 1.66 3  
2 21.01 3.50 3  
3 23.68 3.31 2  
4 24.59 3.61 4

*A seleção com uma ou mais colunas informadas em uma lista, retorna um DataFrame. Note que df['tip'] retorna um pd.Series, enquanto, selecionar df[['tip']] retorna um pd.DataFrame de uma única coluna.*

De modo geral, você pode empreguar a seleção de pd.Series para selecionar um único atributo, e empregar a seleção pd.DataFrame quando estiver interessado na criação de subconjuntos de dados.

## Selecionando Maiores e Menores

Veremos mais adiante como fazer seleção de linhas, mas aqui é oportuno vermos como selecionar linhas que contenham os maiores e os menores valores da coleção.

tips.nsmallest(5,'total\_bill')

total\_bill tip sex smoker day time size  
67 3.07 1.00 Female Yes Sat Dinner 1  
92 5.75 1.00 Female Yes Fri Dinner 2  
111 7.25 1.00 Female No Sat Dinner 1  
172 7.25 5.15 Male Yes Sun Dinner 2  
149 7.51 2.00 Male No Thur Lunch 2

tips\_top3 = tips.nlargest(3,'total\_bill')  
tips\_top3

total\_bill tip sex smoker day time size  
170 50.81 10.00 Male Yes Sat Dinner 3  
212 48.33 9.00 Male No Sat Dinner 4  
59 48.27 6.73 Male No Sat Dinner 4

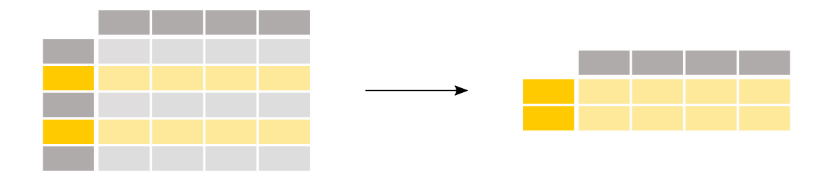
## Alterando e Incluindo Colunas de Dados

Muitas vezes é necessário você criar atributos derivados a partir dos dados originais, como converter os valores para R$, criar um atributo com o valor das contas descontado a gorjeta ou o valor da conta divido pelos participantes da mesa. Essa é uma grande facilidade do Pandas que, a exemplo do NumPy, permite fazer essas operações com um único comando sobre toda uma coleção de dados sem a necessidade de laços de programa.

dolar\_real\_rate = 5.8  
tips['total\_bill'] = tips['total\_bill'] \* dolar\_real\_rate # alterando   
tips['tip'] = tips['tip'] \* dolar\_real\_rate # alterando  
  
tips['total\_bill\_minus\_tips'] = tips['total\_bill'] - tips['tip'] # incluindo um novo atributo  
tips['bill\_by\_head'] = tips['total\_bill'] / tips['size'] # incluindo um novo atributo  
tips.head()

total\_bill tip sex ... size total\_bill\_minus\_tips bill\_by\_head  
0 98.542 5.858 Female ... 2 92.684 49.271000  
1 59.972 9.628 Male ... 3 50.344 19.990667  
2 121.858 20.300 Male ... 3 101.558 40.619333  
3 137.344 19.198 Male ... 2 118.146 68.672000  
4 142.622 20.938 Female ... 4 121.684 35.655500  
  
[5 rows x 9 columns]

## Selecionar Linhas



A seleção de linhas é mais interessante por que podemos especificar condições para os valores que buscamos. Por exemplo, você pode selecionar o tipo de peças que deseja ver em um DataFrame com dados de vários componentes, ou selecionar a as unidades de fábrica de uma certa cidade que você tem os dados de produção.

O critério de seleção é um predicado lógico e você deve empregar a seguinte sintaxe:

df [ <critério de seleção> ]

Exemplos

df [ nome\_coluna == valor ]  
df [ nome\_coluna != valor ]  
df [ nome\_coluna > valor ]

Assim, podemos selecionar os dados somente dos não fumantes ou não fumantes:

tips\_nao\_fumantes = tips[ tips.smoker == 'No' ]  
tips\_fumantes = tips[ tips.smoker == 'Yes' ]

E podemos com isso já responder algumas questões interessantes que envolvem a proporção dos dados de fumantes e não fumantes.

print(f'Percentual de Clientes Fumantes {len( tips\_fumantes ) / len( tips ) :.2f} %')

Percentual de Clientes Fumantes 0.38 %

## Seleção de Linhas e Colunas



O uso mais geral das seleções é quando fazemos seleções de linhas e colunas dos dados, e às vezes nos referimos a esse subconjunto dos dados de *slice* (fatia) dos dados.

A boa prática indica que sempre faremos primeiro a seleção das linhas,

df [ <critério de seleção> ] ...  
  
...Selecionando uma coluna  
df [ <critério de seleção> ].nome\_coluna pd.Series  
df [ <critério de seleção> ]['nome\_coluna']  
  
...Selecionando uma ou mais colunas  
df [ <critério de seleção> ][ [<lista-de-colunas>] ] pd.DataFrame

Com isso podemos fazer inúmeras seleções de dados que temos interesse em visualizar ou analisar, e responder a muitas perguntas sobre os dados.

Fumantes ou Mulheres, pagam mais gorjeta (em média) ?

tips[ tips.smoker == 'Yes' ]['tip'].mean() > tips[ tips.smoker == 'No' ]['tip'].mean()

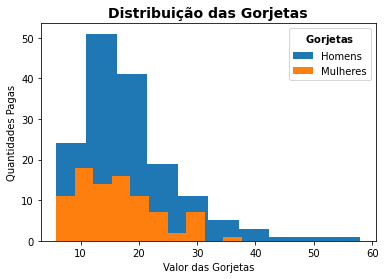
True

tips[ tips.sex == 'Female' ]['tip'].mean() > tips[ tips.sex == 'Male' ]['tip'].mean()

False

As seleções de subconjuntos de dados, com veremos, tem um papel importante na construção de visualizações e entendimento dos dados pois, muitas vezes, estamos interessados em comparar diferentes classes ou subconjuntos dados, como no exemplo abaixo em que observamos os valores das gorjetas de homens e mulheres.

import matplotlib.pyplot as plt  
plt.hist(tips[ tips.sex == 'Male' ]['tip'], label='Homens')  
plt.hist(tips[ tips.sex == 'Female' ]['tip'], label='Mulheres')  
  
plt.xlabel('Valor das Gorjetas')  
plt.ylabel('Quantidades Pagas')  
plt.title('Distribuição das Gorjetas', fontsize=14, weight='bold')  
plt.legend(title='$\\bf{Gorjetas}$')  
plt.show()



O Pandas ainda conta com uma série de operações para manipulação e transformação dos dados como funções de joins e merge de tabelas, groupby etc. empregaremos eventualmente essas operações ao longo do texto e para maiores detalhes você poderá consultar a documentação do Pandas ou outras referências online.

# Agregações de dados

Este é um ponto que não exploramos antes em R, embora existam vários pacotes poderosos para essas e outras manipulações, mas que valerá a pena tratarmos aqui em Python.

Muitas vezes estamos interessados e aplicar uma função ou obter uma estatística dos dados para diferentes *classes* ou grupos de dados simultaneamente. O Pandas oferece uma operação de groupby, que permite manipular e resumir conjuntos de dados dessa forma.

Você já aprendeu que pode aplicar operações como média e máximo para sumarizar estatísticas, por exemplo de uma Serie de dados contendo a idade em um DataFrame de cliente ou o preço do metro quadrado em um DataFramede Imóveis.

Mas como podemos obter as médias de idade por sexo ou o preço máximo do m2 dos imóveis por região?

O groupby é a função que permite agregar esses dados e obter estatísticas ou aplicar funções a várias classes ou grupos de dados.

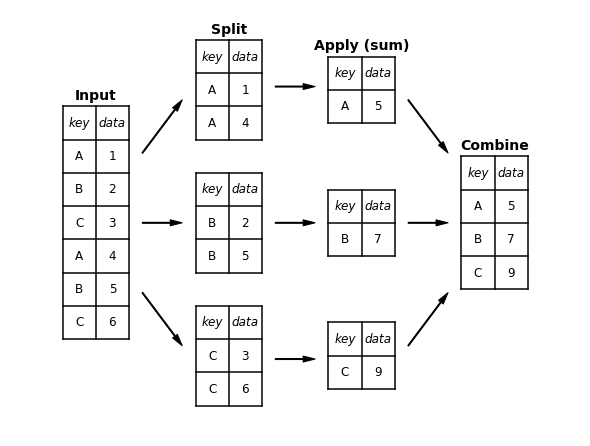
Se você está familiarizado com SQL ou o Excel, o resultado do groupby em Pandas é bastante semelhante ao que você obtém com o groupby do SQL e as tabelas dinâmicas do Excel.

## Funcionamento do GroupBy

A operação de groupby consiste em geral de 3 etapas:

1. Divisão dividir e agrupar dos dados dependendo de um valor de chave especificada.
2. Aplicar uma computação, em geralmente uma estatística agregada como média, *count* ou max, uma transformação ou filtragem para os grupos individuais.
3. Combinar os resultados dessas operações em uma saída única.

# source: https://jakevdp.github.io/PythonDataScienceHandbook/  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
def draw\_dataframe(df, loc=None, width=None, ax=None, linestyle=None,  
 textstyle=None):  
 loc = loc or [0, 0]  
 width = width or 1  
  
 x, y = loc  
  
 if ax is None:  
 ax = plt.gca()  
  
 ncols = len(df.columns) + 1  
 nrows = len(df.index) + 1  
  
 dx = dy = width / ncols  
  
 if linestyle is None:  
 linestyle = {'color':'black'}  
  
 if textstyle is None:  
 textstyle = {'size': 12}  
  
 textstyle.update({'ha':'center', 'va':'center'})  
  
 # draw vertical lines  
 for i in range(ncols + 1):  
 plt.plot(2 \* [x + i \* dx], [y, y + dy \* nrows], \*\*linestyle)  
  
 # draw horizontal lines  
 for i in range(nrows + 1):  
 plt.plot([x, x + dx \* ncols], 2 \* [y + i \* dy], \*\*linestyle)  
  
 # Create index labels  
 for i in range(nrows - 1):  
 plt.text(x + 0.5 \* dx, y + (i + 0.5) \* dy,  
 str(df.index[::-1][i]), \*\*textstyle)  
  
 # Create column labels  
 for i in range(ncols - 1):  
 plt.text(x + (i + 1.5) \* dx, y + (nrows - 0.5) \* dy,  
 str(df.columns[i]), style='italic', \*\*textstyle)  
   
 # Add index label  
 if df.index.name:  
 plt.text(x + 0.5 \* dx, y + (nrows - 0.5) \* dy,  
 str(df.index.name), style='italic', \*\*textstyle)  
  
 # Insert data  
 for i in range(nrows - 1):  
 for j in range(ncols - 1):  
 plt.text(x + (j + 1.5) \* dx,  
 y + (i + 0.5) \* dy,  
 str(df.values[::-1][i, j]), \*\*textstyle)  
  
  
#----------------------------------------------------------  
# Draw figure  
  
import pandas as pd  
df = pd.DataFrame({'data': [1, 2, 3, 4, 5, 6]},  
 index=['A', 'B', 'C', 'A', 'B', 'C'])  
df.index.name = 'key'  
  
  
fig = plt.figure(figsize=(8, 6), facecolor='white')  
ax = plt.axes([0, 0, 1, 1])  
  
ax.axis('off')  
  
draw\_dataframe(df, [0, 0])  
  
for y, ind in zip([3, 1, -1], 'ABC'):  
 split = df[df.index == ind]  
 draw\_dataframe(split, [2, y])  
  
 sum = pd.DataFrame(split.sum()).T  
 sum.index = [ind]  
 sum.index.name = 'key'  
 sum.columns = ['data']  
 draw\_dataframe(sum, [4, y + 0.25])  
   
result = df.groupby(df.index).sum()  
draw\_dataframe(result, [6, 0.75])  
  
style = dict(fontsize=14, ha='center', weight='bold')  
plt.text(0.5, 3.6, "Input", \*\*style)  
plt.text(2.5, 4.6, "Split", \*\*style)  
plt.text(4.5, 4.35, "Apply (sum)", \*\*style)  
plt.text(6.5, 2.85, "Combine", \*\*style)  
  
arrowprops = dict(facecolor='black', width=1, headwidth=6)  
plt.annotate('', (1.8, 3.6), (1.2, 2.8), arrowprops=arrowprops)  
plt.annotate('', (1.8, 1.75), (1.2, 1.75), arrowprops=arrowprops)  
plt.annotate('', (1.8, -0.1), (1.2, 0.7), arrowprops=arrowprops)  
  
plt.annotate('', (3.8, 3.8), (3.2, 3.8), arrowprops=arrowprops)  
plt.annotate('', (3.8, 1.75), (3.2, 1.75), arrowprops=arrowprops)  
plt.annotate('', (3.8, -0.3), (3.2, -0.3), arrowprops=arrowprops)  
  
plt.annotate('', (5.8, 2.8), (5.2, 3.6), arrowprops=arrowprops)  
plt.annotate('', (5.8, 1.75), (5.2, 1.75), arrowprops=arrowprops)  
plt.annotate('', (5.8, 0.7), (5.2, -0.1), arrowprops=arrowprops)  
   
plt.axis('equal')  
plt.ylim(-1.5, 5);  
  
fig.savefig('03.08-split-apply-combine.png')



No exemplo acima,

1. Dividimos os dados nas classes A, B e C
2. Em seguida aplicamos a cada grupo uma função de agregação sobre o valor. No caso aplicamos a sum(), mas poderíamos aplicar outras funções de agregação como count(), mean(), std() etc.
3. Combinamos então esses dados em uma única saída.

Em uma única linha, todas as operação são executadas pela função groupby(),

df.groupby(df.index).sum()

data  
key   
A 5  
B 7  
C 9

Aqui é empregado o índice do DataFrame. Não daremos muita atenção aqui nesta introdução ao Pandas ao uso de índices. Eles tem um papel mais importante no desempenho de algumas operações sobre os dados mas, para os nossos propósitos, é suficiente entendermos os índices como mais um atributo dos dados e que podemos referenciar como df.index.

Outras operações de agregação além do sum() podem ser então igualmente aplicadas a quaisquer atributos desde que façam sentido para os seus valores (posso ter o valor máximo de um atributo do tipo String, mas não um valor de média).

display( df.groupby(df.index).mean() )  
display( df.groupby(df.index).std() )  
display( df.groupby(df.index).min() )

data  
key   
A 2.5  
B 3.5  
C 4.5

data  
key   
A 2.12132  
B 2.12132  
C 2.12132

data  
key   
A 1  
B 2  
C 3

Ou ainda,

display( df.groupby(df.index).agg(['mean', 'std', 'min']) )

data   
 mean std min  
key   
A 2.5 2.12132 1  
B 3.5 2.12132 2  
C 4.5 2.12132 3

Veja, por exemplo, a aplicação em um conjunto de dados mais prático.

pedidos = pd.DataFrame( {'Pedido': pd.Series([ 1, 1, 1, 2, 2, 2, 2, 3, 3]),  
 'Itens': pd.Series([ 'Tênis Corrida', 'Tênis Caminhada', 'Camiseta', 'Tênis Caminhada', 'Camiseta', 'Boné' , 'Keds' ,'Tênis Caminhada', 'Camiseta' ]),  
 'Preco Unidade': pd.Series([ 300, 250, 90, 250, 90, 60, 250, 250, 90]),  
 'Quantidade': pd.Series([ 1, 2, 3, 2, 1, 1, 3, 2, 1]) } )  
pedidos

Pedido Itens Preco Unidade Quantidade  
0 1 Tênis Corrida 300 1  
1 1 Tênis Caminhada 250 2  
2 1 Camiseta 90 3  
3 2 Tênis Caminhada 250 2  
4 2 Camiseta 90 1  
5 2 Boné 60 1  
6 2 Keds 250 3  
7 3 Tênis Caminhada 250 2  
8 3 Camiseta 90 1

Aqui as operações são aplicáveis somente a valores numéricos,

display( pedidos.groupby('Pedido').agg(['mean', 'std', 'min']) )

Preco Unidade Quantidade   
 mean std min mean std min  
Pedido   
1 213.333333 109.696551 90 2.00 1.000000 1  
2 162.500000 101.775898 60 1.75 0.957427 1  
3 170.000000 113.137085 90 1.50 0.707107 1

Mas as funções, se aplicáveis, podem envolver também atributos categóricos,

display( pedidos.groupby('Pedido').max() )

Itens Preco Unidade Quantidade  
Pedido   
1 Tênis Corrida 300 3  
2 Tênis Caminhada 250 3  
3 Tênis Caminhada 250 2

Podemos também selecionar os atributos,

display( pedidos.groupby('Itens')['Quantidade'].agg(['mean', 'std', 'min', 'count']) )  
  
# Experimente aqui, faz pouco sentido obter esses valores para Pedido  
# display( pedidos.groupby('Itens').agg(['mean', 'std', 'min', 'count']) )

mean std min count  
Itens   
Boné 1.000000 NaN 1 1  
Camiseta 1.666667 1.154701 1 3  
Keds 3.000000 NaN 3 1  
Tênis Caminhada 2.000000 0.000000 2 3  
Tênis Corrida 1.000000 NaN 1 1

E, finalmente, as agregações também podem envolver mais que um nível de valores chave,

display( pedidos.groupby( ['Itens','Pedido'] ).agg(['min','max']) )

Preco Unidade Quantidade   
 min max min max  
Itens Pedido   
Boné 2 60 60 1 1  
Camiseta 1 90 90 3 3  
 2 90 90 1 1  
 3 90 90 1 1  
Keds 2 250 250 3 3  
Tênis Caminhada 1 250 250 2 2  
 2 250 250 2 2  
 3 250 250 2 2  
Tênis Corrida 1 300 300 1 1

## Acessando Chaves e Valores dos Grupos

O método groupby retorna um objeto do tipo DataFrameGroupBy e não um DataFrame e essa é uma grande diferença entre essa função no Pandas e no SQL. Os métodos de DataFrame não podem ser aplicados a um obejto do tipo DataFrameGroupBy, enquanto na SQL um SELECT com GROUPY no SQL retorna uma tabela e todas as operações de tabela são aplicáveis.

pedidos.groupby('Itens')

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f58f17ccdd0>

A estrutura desses dados é basicamente na forma de um dicionário e você pode acessar as chaves e os seus valores.

grupos\_itens = pedidos.groupby('Itens')  
print ( grupos\_itens )  
print ( grupos\_itens.groups )  
print ( grupos\_itens.groups.keys() )  
print ( grupos\_itens.groups.values() )

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f58f15a0390>  
{'Boné': [5], 'Camiseta': [2, 4, 8], 'Keds': [6], 'Tênis Caminhada': [1, 3, 7], 'Tênis Corrida': [0]}  
dict\_keys(['Boné', 'Camiseta', 'Keds', 'Tênis Caminhada', 'Tênis Corrida'])  
dict\_values([Int64Index([5], dtype='int64'), Int64Index([2, 4, 8], dtype='int64'), Int64Index([6], dtype='int64'), Int64Index([1, 3, 7], dtype='int64'), Int64Index([0], dtype='int64')])

### Iterando sobre grupos

Você pode, então, fazer iterações sobre esses valores ou aplicar funções de agregação,

for chave in pedidos.groupby('Itens').groups.keys():  
 print(chave)  
  
# que é o mesmo que,  
#  
# for chave in grupos\_itens.groups.keys():  
# print(chave)  
  
for chave, grupo in pedidos.groupby('Itens'):  
 print(chave)  
 print(grupo)

Boné  
Camiseta  
Keds  
Tênis Caminhada  
Tênis Corrida  
Boné  
 Pedido Itens Preco Unidade Quantidade  
5 2 Boné 60 1  
Camiseta  
 Pedido Itens Preco Unidade Quantidade  
2 1 Camiseta 90 3  
4 2 Camiseta 90 1  
8 3 Camiseta 90 1  
Keds  
 Pedido Itens Preco Unidade Quantidade  
6 2 Keds 250 3  
Tênis Caminhada  
 Pedido Itens Preco Unidade Quantidade  
1 1 Tênis Caminhada 250 2  
3 2 Tênis Caminhada 250 2  
7 3 Tênis Caminhada 250 2  
Tênis Corrida  
 Pedido Itens Preco Unidade Quantidade  
0 1 Tênis Corrida 300 1

pedidos.groupby('Itens').Quantidade.sum() # total de quantidades por item  
  
# que é o mesmo que,  
# grupos\_itens.Quantidade.sum() # total de quantidades por item

Itens  
Boné 1  
Camiseta 5  
Keds 3  
Tênis Caminhada 6  
Tênis Corrida 1  
Name: Quantidade, dtype: int64

### Acessando os valores

Você pode referenciar diretamente o atributo em um grupo para obter uma Serie Pandas e, então, operar esses valores como abaixo,

pedidos.groupby('Itens')[['Quantidade','Preco Unidade']].sum()

Quantidade Preco Unidade  
Itens   
Boné 1 60  
Camiseta 5 270  
Keds 3 250  
Tênis Caminhada 6 750  
Tênis Corrida 1 300

pedidos.groupby('Itens')[['Quantidade','Preco Unidade']].sum()['Quantidade']

Itens  
Boné 1  
Camiseta 5  
Keds 3  
Tênis Caminhada 6  
Tênis Corrida 1  
Name: Quantidade, dtype: int64

type( pedidos.groupby('Itens')[['Quantidade','Preco Unidade']].sum()['Quantidade'] )

pandas.core.series.Series

pedidos.groupby('Itens')[['Quantidade','Preco Unidade']].sum()['Quantidade'].mean()

3.2

Mas um artifício bastante prático que você pode empregar, se não estiver preocupado com o desempenho e uso de recursos de memória das operações, é transformar o seu agrupamento em um DataFrame para que você possa aplicar as operações que já conhece para essa classe diretamente,

df = pd.DataFrame( pedidos.groupby('Itens')[['Quantidade','Preco Unidade']].sum() ).reset\_index()  
df.head()

Itens Quantidade Preco Unidade  
0 Boné 1 60  
1 Camiseta 5 270  
2 Keds 3 250  
3 Tênis Caminhada 6 750  
4 Tênis Corrida 1 300

***Outras Transformações dos Dados.*** *Muitas vezes a Análise dos Dados requer uma Preparação dos Dados. Você pode entender isso como uma série de transformações sobre os dados que buscam torná-los mais adequados para análise. As seleções e agrupamentos que vimos aqui são parte dessa preparação. Mas elas basicamente reduzem os dados a uma coleção de interesse e existem muitas outras transformações úteis e importantes. Além das seleções essas transformações podem ser de vários tipos:*

* ***Transformações de formato*** *(datas, unidades etc.)*
* ***Transformações de tipo*** *(discretização de valores, transformação de dados categóricos para numéricos etc.)*
* ***Tratamento de dados faltantes*** *(exclusão, imputação de dados etc.)*
* ***Inclusão de atributos derivados*** *(sub cadeias de caracteres de um atributo, transformação de uma data no dia da semana ou de uma localização geográfica em coordenadas latitude/longitude etc.)*
* ***Normalizações de valores*** *(no tratamento de dados com diferentes escalas)*
* ***Merge e Junção de dados de diferentes fontes*** *(à exemplo dos joins e união de tabelas da SQL em Banco de Dados)*
* ***Reshape dos dados*** *(transformação do formato dos dados, como transformar linhas em colunas ou vice-versa)*

*Isso só para citar as principais. Essas transformações são todas possíveis tanto em R como em Python, mas serão tratadas mais adiante no curso. Aqui nos limitaremos a fazer seleções e agrupamentos como acima. Operações simples de transformação podem ser consultadas nos manuais e referências, tanto de Python como de R e, em alguns casos, estaremos fazendo uso de algumas dessas funções aqui em casos mais simples.*

# Visualização dos Dados, Matplotlib

Agora que você já sabe selecionar dados de interesse à partir de um conjunto inicial vamos nos voltar para a visualização desses dados. Aqui você vai aprender a produzir visualizações dos dados principalmente empregando a biblioteca Matplotlib e, mais adiante também o pacote Seaborn. Nossa escolha pelo Matplotlib é que ela é a principal biblioteca Python para produzir visualizações de dados. Pacotes mais recentes e com mais poder para a exibição dos dados, como o Seaborn, Plotly e Altair, são construídos sobre o Matplotlib e, assim, compartilham boa parte dos comandos do Matplotlib que aprenderemos aqui.

O Matplotlib é uma biblioteca bastante completa e permite fazer tanto visualizações de dados de **Exploração dos Dados** como de **Apresentação**, que vimos na aula 4. Assim como nas visualizações em R, nosso foco aqui será a construção de visualizações para a exploração dos dados, muito mais do que para construção de apresentação. De qualquer modo, já tendo passado por alguns conceitos de visualização de dados em R, podemos nos aprofundar um pouco mais nos ornamentos das visualizações em Python, como títulos, emprego de cores e diferentes estilos de linha, rótulos etc. Esses ornamentos, em muitos casos, não são meramente um luxo ou um recurso estético, eles enriquecem e permitem comunicar a sua Análise de forma mais efetiva o que é essencial na parte dos casos práticos (projetos, grupos de trabalho, empresas etc.).

Vamos apresentar os gráficos em Python organizados por sua finalidade (gráficos de evolução, distribuição etc.) do mesmo modo que organizamos os gráficos em R. Mas antes precisamos aprender uma série de comandos básicos do Matplotlib que podem ser aplicados de forma geral a quaisquer visualizações.

# Gráficos em Matplotlib

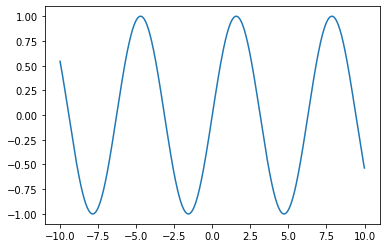
O pacote Matplotlib é uma biblioteca gráfica para visualização de dados em Python. O Seaborn é uma biblioteca de visualização de dados Python construída sobre Matplotlib que fornece uma interface de mais alto nível para desenhar gráficos e possui muitos recursos atraentes em especial para a análise de dados multivariados (com muitas dimensões, isto é, atributos, que desejamos analisar em conjunto). Em alguns casos empregaremos essa biblioteca e por isso faremos sempre um conjunto básico de imports incluindo ambas as bibliotecas.

# imports empregados nesta seção. Eles podem se repetir abaixo para enfatizar os recursos.  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
%matplotlib inline  
import seaborn as sns

## Um gráfico de linha simples plt.plot(x,y)

Os gráficos de linha são os gráficos mais simples e, antes explorarmos outros tipos de gráficos, vamos entender como criar gráficos de linha com o Matplotlib. Eles ainda são particularmente úteis para representar gráficos de funções.

import numpy as np  
  
import matplotlib.pyplot as plt   
%matplotlib inline  
  
x = np.arange(-10,10,0.01)  
  
plt.plot(x, np.sin(x))  
plt.show()



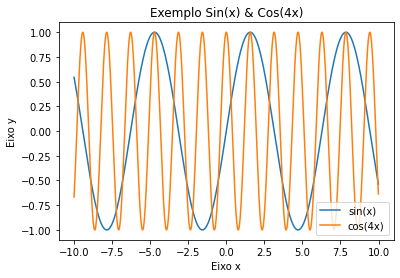
Acima você encontra primeiramente o import do pacote Matplotlib.

Você pode empregar o Matplotlib em um script Python, em um terminal Python interativo (IPython) ou em um Python notebook. Todas as execuções aqui serão feitas em Python notebooks e ao empregarmos o comando %matplotlib inline estamos dizendo ao backend de execução do Python que os gráficos produzidos pelo Matplotlib serão incluídos em seu notebook, juntamente com o código.

A função show() concluí a exibição de um gráfico. Ela atua com o backend gráfico interativo do sistema e os detalhes de suas operações variam conforme o contexto de execução. Basicamente ela procura os objetos ativos para a construção da figura abre uma ou mais janelas para exibir a saída dos gráficos. No caso de uma execução em Python notebook, ela exibe a figura abaixo da célula de código sem a numeração da célula out[] (para Jupyter notebooks) ou texto de referência do objeto construído (Colab). Ela aparece no final dos scripts de comandos gráficos e uma única vez.

Você ainda pode adicionar títulos, rótulos aos eixos, legendas e alterar propriedades da linha como estilo e cor.

import numpy as np  
  
import matplotlib.pyplot as plt   
%matplotlib inline  
  
x = np.arange(-10,10,0.01)  
  
plt.plot(x, np.sin(x), label='sin(x)')  
plt.plot(x, np.cos(4\*x), label='cos(4x)')  
  
plt.title('Exemplo Sin(x) & Cos(4x)')  
plt.xlabel('Eixo x')  
plt.ylabel('Eixo y')  
plt.legend()  
  
plt.show()



Note que gráficos das duas funções acimas são sobrepostos. Ao executar o comando plot existe uma única 'figura' ativa, e todos os comandos, como título, legendas e rótulos, são direcionados do mesmo modo que o plot para a mesma figura. A função show() se encarrega depois de combinar todos objetos ativos em uma única figura. Experimente incluir o plt.show() após o primeiro plot e veja o resultado.

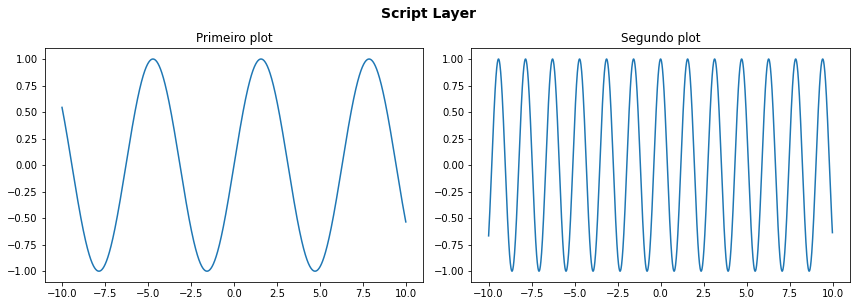
## Empregando a interface direta ou de objetos Script Layer & Artist Layer

Os comandos em Matplotlib podem ser empregados de duas formas. Na interface direta (ou Script Layer) o Matplotlib mantém o controle da figura e eixos 'atuais' onde as funções plt serão aplicadas. Essa é uma forma simples e útil, e vamos usá-la na maior parte do tempo. É, entretanto, uma forma complicada quando trabalhamos com múltiplos gráficos ou conjuntos de gráficos com uma apresentação muito complexa.

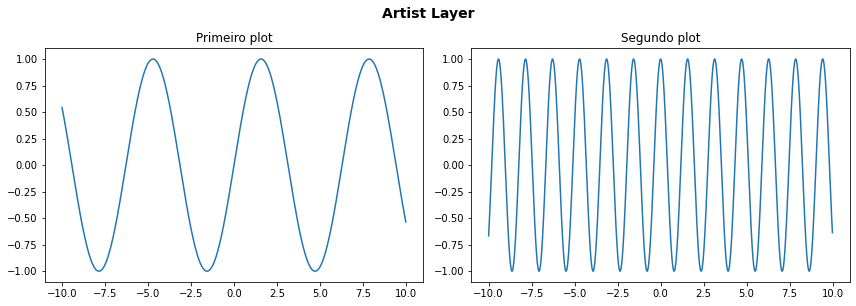
A interface orientada a objetos (ou Artist Layer) permite um maior controle das figuras e você deve preferir essa interface quando múltiplos gráficos ou figuras muito elaboradas são empregados. Nela você pode apontar explicitamente a 'figura' (fig) os gráficos contidos dentro dela (ax ou axis dependendo do caso).

Compare os gráficos abaixo. O primeiro emprega a interface direta. Em uma figura com múltiplos gráficos, cada novo gráfico é ativado (subplot(linhas, colunas, gráfico ativo)) e todos os comandos plt são então direcionados para o gráfico ativo. A execução seguinte emprega a interface orientada a objetos e os comandos plot são direcionados explicitamente para cada gráfico (ax[i]).

plt.figure(figsize=(12,4)) # cria uma figura alterando o tamanho, figsize=(tamanho\_horizontal,tamanho\_vertical)  
  
plt.suptitle('Script Layer', fontsize=14, weight='bold', y=1.05)  
  
# cria e ativa um gráfico na figura  
plt.subplot(1, 2, 1) # (linhas, colunas, gráfico ativo)  
plt.plot(x, np.sin(x))  
plt.title('Primeiro plot')  
  
# cria e ativa um segundo gráfico na figura  
plt.subplot(1, 2, 2) # (linhas, colunas, gráfico ativo)  
plt.plot(x, np.cos(4\*x));  
plt.title('Segundo plot')  
  
plt.tight\_layout()  
plt.show()



# Cria uma grade de gráficos com 1 linha e 2 colunas de gráficos, ax é um vetor em que cada elemento corresponde a um gráfico da grade  
fig, ax = plt.subplots(1,2,figsize=(12,4)) # cria uma figura alterando o tamanho, figsize=(tamanho\_horizontal,tamanho\_vertical)  
  
plt.suptitle('Artist Layer', fontsize=14, weight='bold', y=1.05)  
  
# Os comandos podem então ser direcionados para o gráfico específico  
ax[0].plot(x, np.sin(x))  
ax[0].set\_title('Primeiro plot')  
  
ax[1].plot(x, np.cos(4\*x));  
ax[1].set\_title('Segundo plot')  
  
plt.tight\_layout()  
plt.show()



Mais adiante você verá melhor como criar múltiplos gráficos. Por hora é suficiente você entender que existem essas duas formas de trabalho com o Matplotlib.

## Propriedades de linha linestyle, marker, color, linewidth e markersize

Você pode alterar várias das propriedades das linhas dos gráficos como estilo da linha, cor e sua espessura. Isso é importante para diferenciar várias linhas em um mesmo gráfico ou ainda se você quiser dar destaque a uma ou mais linhas.

Essas propriedades podem ser codificadas explicitamente:

plt.plot(x, np.cos(x), linestyle='dashed', marker='o', color='green', linewidth=1, markersize=6)

Ou de forma resumida:

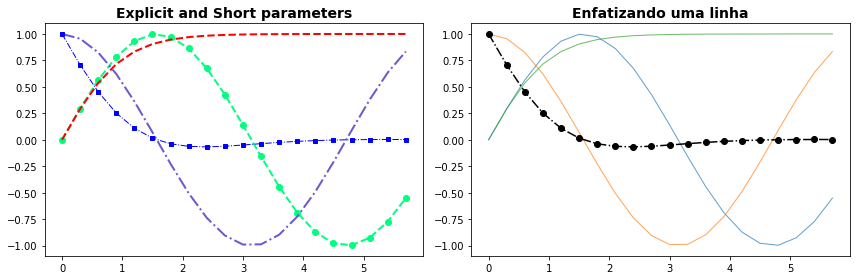
plt.plot(x, np.cos(x), '--og', linewidth=1, markersize=6)

A expressão '--og' agrupa os parâmetros --, o e g e indicam respectivamente os mesmos parâmetros que o modo explícito linestyle='dashed', marker='o', color='green'. Mesmo na forma resumida eles podem aparecer em qualquer ordem, uma vez que os símbolos de cada parâmetro não se repetem. As cores nesse modo ficam entretanto limitadas ao um conjunto de cores básicas rgbcmyk (r=red, g=green, ...).

O esquema de cores é bastante sofisticado e permite empregar de cores básicas à tons de cinza, como também especificar valores de cada componente RGB (esquema de combinação de cores básicas Red, Green, Blue) e transparência de linhas (parâmetro alpha).

A seguir o primeiro gráfico mostra como empregar o uso dos parâmetros explícitos e de forma resumida, e o gráfico seguinte mostra como essas configurações podem ser empregadas para dar destaque a uma ou mais linhas em um gráfico.

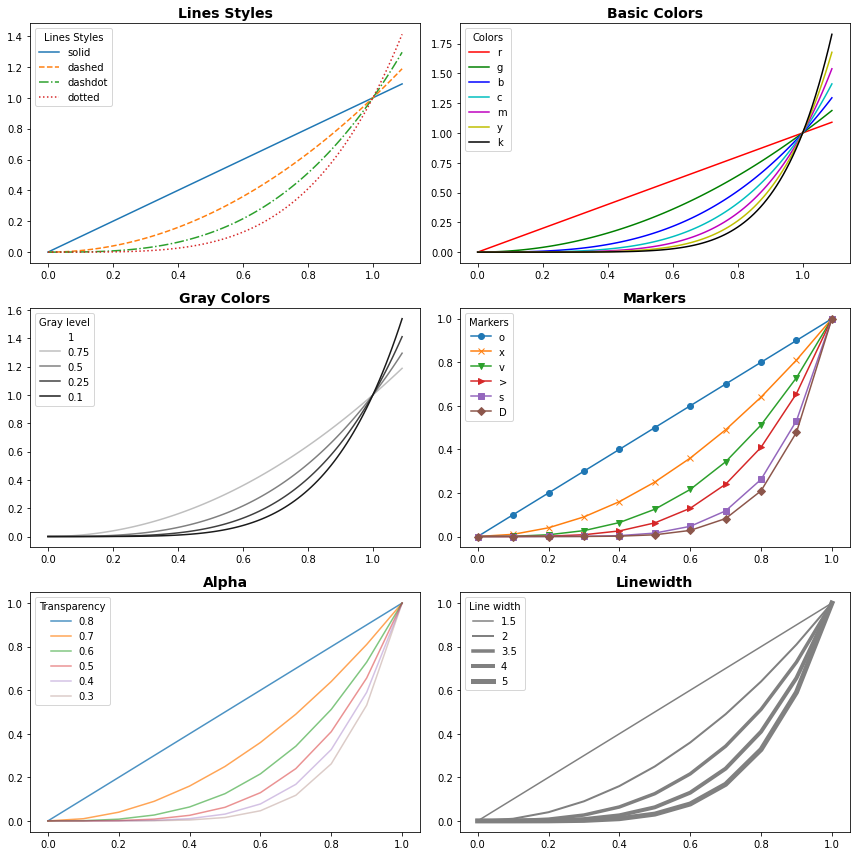
x = np.arange(0, 6, 0.3)  
  
fig = plt.figure(figsize=(12,4))  
  
plt.subplot(1, 2, 1)  
plt.title('Explicit and Short parameters', fontsize=14, weight='bold')  
plt.plot(x, np.sin(x), color='springgreen', marker='o', linestyle='dashed', linewidth=2, markersize=6)  
plt.plot(x, np.cos(x), color='slateblue', linestyle='dashdot', linewidth=2)  
  
# short color code (rgbcmyk)  
plt.plot(x, np.tanh(x), '--r', linewidth=2, markersize=6)  
plt.plot(x, 1/np.exp(x)\*np.cos(x), '-.sb', linewidth=1, markersize=4)  
  
plt.subplot(1, 2, 2)  
plt.title('Enfatizando uma linha', fontsize=14, weight='bold')  
plt.plot(x, np.sin(x), linewidth=1, alpha=0.7)  
plt.plot(x, np.cos(x), linewidth=1, alpha=0.7)  
plt.plot(x, np.tanh(x), linewidth=1, alpha=0.7)  
plt.plot(x, 1/np.exp(x)\*np.cos(x), '-.ok')  
  
plt.tight\_layout()  
plt.show()



*Você pode empregar diferentes estilos de linha apenas para diferenciar as linhas em gráficos de múltiplos, mas também pode empregar o recurso para destacar a uma ou mais linhas do gráfico.*

Abaixo você também encontra alguns exemplos de estilos de linha, marcadores e cores básicas mais empregados. Eles serão suficientes aqui e você pode consultar a documentação do Matplotlib para uma relação completa desses parâmetros.

x = np.arange(0.0, 1.1, 0.01)  
  
fig = plt.figure(figsize=(12,12))  
  
plt.subplot(3, 2, 1)  
plt.title('Lines Styles', fontsize=14, weight='bold') # automatic colors  
plt.plot(x, x\*\*1, linestyle='solid', label='solid')  
plt.plot(x, x\*\*2, linestyle='dashed', label='dashed')  
plt.plot(x, x\*\*3, linestyle='dashdot', label='dashdot')  
plt.plot(x, x\*\*4, linestyle='dotted', label='dotted')  
plt.legend(title='Lines Styles')  
  
plt.subplot(3, 2, 2)  
plt.title('Basic Colors', fontsize=14, weight='bold')  
plt.plot(x, x\*\*1, color='r', label='r')  
plt.plot(x, x\*\*2, color='g', label='g')  
plt.plot(x, x\*\*3, color='b', label='b')  
plt.plot(x, x\*\*4, color='c', label='c')  
plt.plot(x, x\*\*5, color='m', label='m')  
plt.plot(x, x\*\*6, color='y', label='y')  
plt.plot(x, x\*\*7, color='k', label='k')  
plt.legend(title='Colors')  
  
plt.subplot(3, 2, 3)  
plt.title('Gray Colors', fontsize=14, weight='bold')  
plt.plot(x, x\*\*1, color='1', label='1')  
plt.plot(x, x\*\*2, color='0.75', label='0.75')  
plt.plot(x, x\*\*3, color='0.5', label='0.5')  
plt.plot(x, x\*\*4, color='0.25', label='0.25')  
plt.plot(x, x\*\*5, color='0.1', label='0.1')  
plt.legend(title='Gray level')  
  
x = np.arange(0.0, 1.1, 0.1) # para melhor exibição dos dados com markers  
  
plt.subplot(3, 2, 4)  
plt.title('Markers', fontsize=14, weight='bold')  
plt.plot(x, x\*\*1, marker='o', label='o')  
plt.plot(x, x\*\*2, marker='x', label='x')  
plt.plot(x, x\*\*3, marker='v', label='v')  
plt.plot(x, x\*\*4, marker='>', label='>')  
plt.plot(x, x\*\*6, marker='s', label='s')  
plt.plot(x, x\*\*7, marker='D', label='D')  
plt.legend(title='Markers')  
  
# o parâmetro alpha permite configurar um grau de transparência nas cores de linha  
plt.subplot(3, 2, 5)  
plt.title('Alpha', fontsize=14, weight='bold')  
plt.plot(x, x\*\*1, alpha=0.8, label='0.8')  
plt.plot(x, x\*\*2, alpha=0.7, label='0.7')  
plt.plot(x, x\*\*3, alpha=0.6, label='0.6')  
plt.plot(x, x\*\*4, alpha=0.5, label='0.5')  
plt.plot(x, x\*\*5, alpha=0.4, label='0.4')  
plt.plot(x, x\*\*6, alpha=0.3, label='0.3')  
plt.legend(title='Transparency')  
  
# empregando diferente espessuras de linha  
plt.subplot(3, 2, 6)  
plt.title('Linewidth', fontsize=14, weight='bold')  
plt.plot(x, x\*\*1, color='0.5', linewidth=1.5, label='1.5')  
plt.plot(x, x\*\*2, color='0.5', linewidth=2, label='2')  
plt.plot(x, x\*\*3, color='0.5', linewidth=3.5, label='3.5')  
plt.plot(x, x\*\*4, color='0.5', linewidth=4, label='4')  
plt.plot(x, x\*\*5, color='0.5', linewidth=5, label='5')  
plt.legend(title='Line width')  
  
plt.tight\_layout()  
plt.show()



## Textos: title, labels, lengend e text

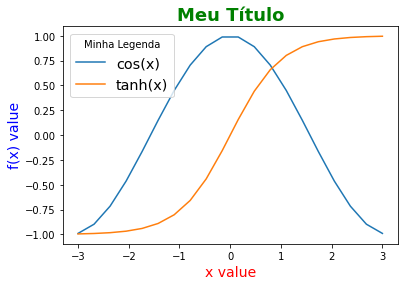
Textos são um complemento importante de qualquer gráfico. Sem eles você não tem como saber exatamente ao que se refere cada eixo, sua escala ou sua unidade de medida. O título ainda pode explicitar a mensagem que você deseja passar e anotações ainda podem ajudar a indicar pontos específicos do gráfico que você deseja destacar.

O Matplotlib, como outros pacotes de visualização, permite adicionar inúmeras anotações aos gráficos como títulos, rótulos ou labels para as linhas, nomear os eixos ou até mesmo inserir uma anotação em uma posição específica do gráfico. Isso é muito importante para tornar claras as informações exibidas.

*Textos como rótulos dos eixos e outras anotações* ***não são opcionais****. A menos que conheça o contexto, não tem como você saber se um gráfico está apresentando a evolução de uma ação na bolsa no último ano ou a mudança de temperatura global na última década!*

O exemplo a seguir mostra como definir um título para o seu gráfico, nomear os eixos e como inserir rótulos (labels) para cada uma das linhas de um gráfico que emprega várias linhas.

x = np.linspace(-3, 3, 20)  
  
plt.figure()  
  
# Adicionando labels às diferentes linhas do gráfico  
plt.plot(x, np.cos(x), label='cos(x)')  
plt.plot(x, np.tanh(x), label='tanh(x)')  
  
# Adicionando o Título  
plt.title('Meu Título', color='green', fontsize=18, weight='bold', ha='center')  
  
# Nomeando os eixos  
plt.xlabel('x value', color='red', fontsize=14)  
plt.ylabel('f(x) value', color='blue', fontsize=14)  
  
# Exibindo a legenda  
plt.legend(title='Minha Legenda', loc='best', fontsize='x-large')  
  
plt.show()



Títulos e rótulos aceitam todas as configurações comuns de editores de texto como tamanhos de fonte (fontsize), cor (color), estilos (como por exemplo weight='bold' e style='italic'). Para tipos de fonte diferentes é necessário modificar parâmetros à partir do plt.rcParams e você pode querer experimentar por exemplo plt.rcParams["font.family"] = "fantasy" ou um outro tipo de fonte. Aqui manteremos sempre o uso da fonte padrão.

Os títulos podem ser alinhados no centro (padrão), à esquerda ou à direita usando o parâmetro ha (horizontalalignment, e valores 'center','right' ou 'left') e você pode deixar para o Matplotlib definir a melhor posição da legenda ou empregar o parâmetro de localização loc (loc='upper right', loc='lower left', loc='center left' etc.).

Quando trabalhando com múltiplos gráficos pode ser necessário adicionar também um suptitle ao conjunto de gráficos e empregamos isso em um exemplo anterior quando apresentamos o modo Artist Layer:

plt.suptitle('Artist Layer', fontsize=14, weight='bold', y=1.05)

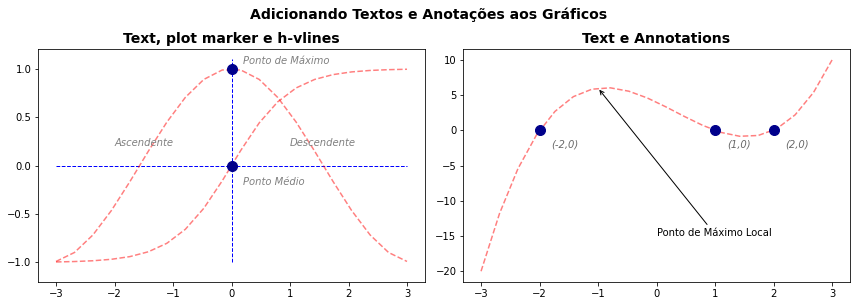
O y=1.05 coloca o suptitle 0.05 acima da figura e evita a sobre posição dos títulos. O comando plt.tight\_layout(), no mesmo exemplo, ajusta o automaticamente o layout da figura para evitar sobreposições dos gráficos, como por exemplo dos textos dos eixos de um gráfico com outro e vamos empregar bastante esse comando aqui.

*Anotações dentro de um gráfico são essencias quando se quer chamar a atenção do leitor para um determinado ponto do gráfico.*

Para isso o Matplotlib fornece duas funções, o plt.text() e o plt.annotate(). O primeiro simplesmente adiciona um texto em uma posição x, y, o segundo, insere o texto adicionando uma seta do texto a um ponto do gráfico.

Pontos específicos no gráfico podem ainda ser destacados, ou com o comando plot() destacando um ponto em coordenada específica, ou com a adição de linhas verticais (vlines()) ou horizontais (hlines()) informando a posição e os limites mínimo e máximo de cada linha.

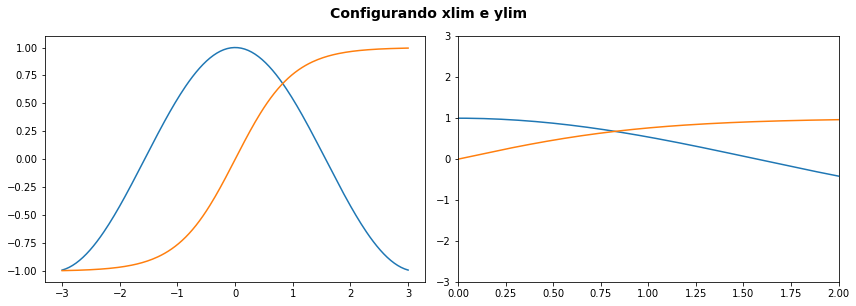
x = np.linspace(-3, 3, 20)  
  
fig = plt.figure(figsize=(12,4))  
plt.suptitle('Adicionando Textos e Anotações aos Gráficos', fontsize=14, weight='bold', y=1.05)  
  
plt.subplot(1, 2, 1)  
plt.title('Text, plot marker e h-vlines', fontsize=14, weight='bold')  
plt.plot(x, np.tanh(x), 'r--', alpha=0.5)  
plt.plot(x, np.cos(x), 'r--', alpha=0.5)  
plt.ylim([-1.2,1.2])  
  
# Enfatizando um ponto específico do gráfico  
plt.plot(0, 1, marker = 'o' , color = 'darkblue', markersize= 10)  
plt.plot(0, 0, marker = 'o' , color = 'darkblue', markersize= 10)  
  
# Adicionando anotações  
plt.text(0 + 0.2, 0 - 0.2, 'Ponto Médio', style='italic', color='grey')  
plt.text(0 + 0.2, 1 + 0.05, 'Ponto de Máximo', style='italic', color='grey')  
plt.text(-2.0, 0.2, 'Ascendente', style='italic', color='grey')  
plt.text(1, 0.2, 'Descendente', style='italic', color='grey')  
  
# Adicionando linhas verticais e horizontais  
plt.vlines(0, -1, 1.1, linestyle='dashed', color='b', linewidth=1) # vlines(posição, início, fim)  
plt.hlines(0, -3, 3, linestyle='dashed', color='b', linewidth=1) # hlines(posição, início, fim)  
  
plt.subplot(1, 2, 2)  
plt.title('Text e Annotations', fontsize=14, weight='bold')  
plt.plot(x, (x + 2) \* (x - 1) \* (x - 2), 'r--', alpha=0.5)  
  
# Enfatizando vários pontos do gráfico  
plt.plot([-2, 1, 2], [0, 0, 0], 'o', color='darkblue', markersize=10)  
  
# Adicionando anotações  
plt.text(-2 + 0.2, 0 - 2.4, '(-2,0)', style='italic', color='dimgrey')  
plt.text(1 + 0.2, 0 - 2.4, '(1,0)', style='italic', color='dimgrey')  
plt.text(2 + 0.2, 0 - 2.4, '(2,0)', style='italic', color='dimgrey')  
  
# Annotate  
plt.annotate('Ponto de Máximo Local',   
 xy=(-1, 6), # ponto do gráfico  
 xytext=(0, -15), # posição do texto  
 arrowprops=dict(arrowstyle="->"))   
  
plt.tight\_layout()   
plt.show()



## Limite dos eixos e escalas xlim e ylim

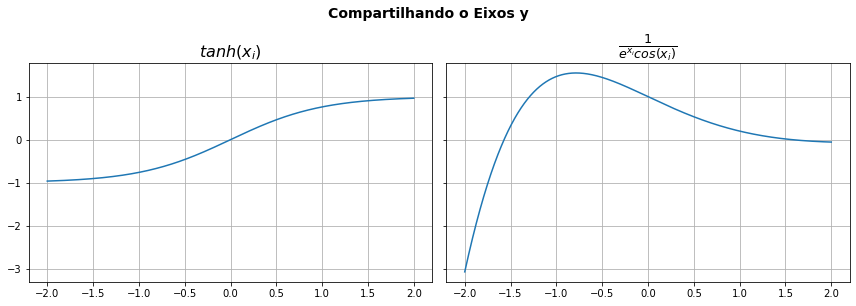
O Matplotlib ajusta automaticamente os limites dos valores dos eixos x e y e as escalas empregadas. Muitas vezes, entretanto, esses valores não são os mais adequados e você pode querer alterá-los. Os limites dos valores podem ser definidos empregando as funções xlim e ylim.

x = np.linspace(-3, 3, 200)  
  
fig = plt.figure(figsize=(12,4))  
plt.suptitle('Configurando xlim e ylim', fontsize=14, weight='bold', y=1.05)  
  
plt.subplot(1,2,1)  
plt.plot(x, np.cos(x))  
plt.plot(x, np.tanh(x))  
  
plt.subplot(1,2,2)  
plt.plot(x, np.cos(x))  
plt.plot(x, np.tanh(x))  
plt.xlim([0,2])  
plt.ylim([-3,3])  
  
plt.tight\_layout()  
plt.show()



Ao trabalhar múltiplos gráficos em uma figura podemos querer compartilhar as escalas dos eixos x e y deixando mais limpa e elegante a exibição dos gráficos. Para isso você pode empregar os parâmetros sharex ou sharey da função subplots conforme a escala do eixo que deseja compartilhar. No exemplo abaixo ainda inserimos uma 'grade' ao gráfico apresentando, assim, mais esse recurso.

fig, ax = plt.subplots(1,2,figsize=(12,4), sharey=True, sharex=True)  
plt.suptitle('Compartilhando o Eixos y', fontsize=14, weight='bold', y=1.05)  
  
# Mais pontos gerados para exibir diferentes réguas de dados nos eixos  
x = np.linspace(-2, 2, 10000)  
  
ax[0].set\_title('$tanh(x\_i)$', fontsize=16)  
ax[0].plot(x, np.tanh(x))  
ax[0].grid()  
  
ax[1].set\_title('$\\frac{1}{e^{x\_i} cos(x\_i)}$', fontsize=18)  
ax[1].plot(x, 1/np.exp(x)\*np.cos(x))  
ax[1].grid()  
  
plt.tight\_layout()  
plt.show()



Talvez você tenha notado que nos dois últimos exemplos, diferente dos exemplos anteriores, empregamos a interface orientada a objetos no lugar da interface direta e que comando para inserir o título foi ligeiramente diferente da forma que empregamos antes.

*Ao empregar a interface orientada a objetos as funções para se definir limites, rótulos e títulos diferem ligeiramente das funções empregadas na interface direta.*

A maior parte das funções plt pode ser aplicada diretamente como métodos de eixo (por exemplo plt.plot() → ax.plot()) mas os comandos para definir limites, rótulos e títulos diferem ligeiramente quando empregados na interface orientada a objeto e adiciona-se o prefixo set\_ aos comandos:

plt.title() → ax.set\_title()

plt.xlabel() → ax.set\_xlabel()

plt.ylabel() → ax.set\_ylabel()

plt.xlim() → ax.set\_xlim()

plt.ylim() → ax.set\_ylim()

E o comando ax.set() ainda pode ser empregado para definir todas essas propriedades simultaneamente.

## LaTex

No exemplo anterior criamos títulos com expressões matemáticas empregando o texto entre símbolos $. Se você está familiarizado com a notação LaTex para símbolos e expressões matemáticas você poderá empregar essa notação também nos textos do Matplotlib e é particularmente elegante para expressões que envolvem símbolos matemáticos como , , , , , , funções como , , , ou ainda expressões mais complexas como:

## Empregando Estilos

Como muitos outros pacotes de visualização o Matplotlib implementa um padrão de visualização dos dados como cores, estilos de linha, fontes que você pode alterar se quiser. Isso é muito útil porque permite que você não precise ficar definindo todas essas configurações, podendo adotar a configuração padrão e alterando somente os elementos que são de interesse. De fato, muitas vezes, empregamos somente a configuração padrão para produzir gráficos rapidamente para, em seguida, modificar os parâmetros conforme a nossa necessidade.

Mas o Matplotlib ainda implementa vários *estilos* predefinidos com padrões diferentes que você pode escolher.

print(plt.style.available)

['Solarize\_Light2', '\_classic\_test\_patch', 'bmh', 'classic', 'dark\_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid', 'tableau-colorblind10']

Por exemplo, o estilo 'ggplot' emula a estética do ggplot, um importante pacote de visualização da linguagem R. Para usar esse estilo, basta você empregar:

plt.style.context('ggplot')

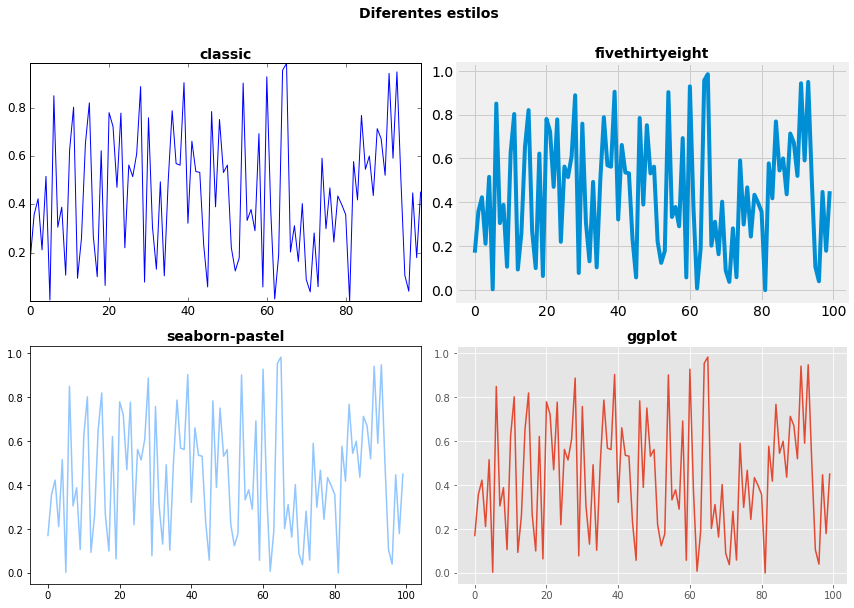
para todos os gráficos da sua sessão, ou ainda:

with plt.style.context('stylename'):  
 <seu plot>

para configurar temporariamente um estilo.

Você pode ver abaixo gráficos com diferentes estilos padrão do Matplotlib, e pode modificar o código para experimentar outros.

# apenas a produção dos dados  
x = np.random.random\_sample(100)  
  
plt.figure(figsize=(12,8))   
plt.suptitle('Diferentes estilos', fontsize=14, weight='bold', y=1.05)  
  
i = 1  
for style in ['classic','fivethirtyeight','seaborn-pastel','ggplot']:  
 with plt.style.context(style):  
 plt.subplot(2, 2, i)   
 plt.plot(x)   
 plt.title(style, fontsize=14, weight='bold')   
 i = i + 1   
  
plt.tight\_layout()  
plt.show()  
  
plt.style.context('default')  
plt.show()

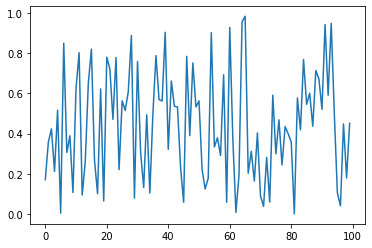


## Salvando os Gráficos em um Arquivo

Salvar os gráficos para uso em algum outro aplicativo ou documento, como uma página web, um relatório ou uma apresentação é muito comum. O comando savefig() permite salvar seus gráficos em uma série de formatos como .JPG, .PNG, .PDF ou .SVG. e o parâmetro savefig.dpi dever ser empregado para aumentar a resolução e qualidade da imagem produzida.

*Empregue plt.savefig( <arquivos>.JPG|PNG|PDF|SVG ) para salvar gráficos em arquivos no seu diretório de trabalho do Python.*

plt.rcParams['savefig.dpi'] = 300  
  
plt.plot(x)  
  
plt.savefig('my\_plot.png')  
plt.savefig('my\_plot.pdf')  
  
plt.show()



# Tipos de Gráficos

Vamos lembrar aqui o que aprendemos ao estudar visualizações de dados em R (aula 4) sobre o tipo de perguntas que podemos fazer sobre os dados e os principais gráficos envolvidos:

* **Evolução** (ou *Tendência dos Dados*): gráficos de linha, área, séries múltiplas
* **Distribuição**: histogramas, gráficos de distribuição de densidade, *boxplot*
* **Quantidades e Proporções**: gráficos de barras, *word cloud*, *spider*, *pie chart*
* **Relações**: gráficos de dispersão, *heat map*, *density 2D*

Com o instrumental básico de Python que vimos até aqui você pode começar a explorar diferentes dados e vamos apresentar os gráficos em Python organizados da mesma maneira.

# Evolução

Vamos lembrar que quando falamos de Evolução dos dados estamos interessados em observar a evolução de um conjunto de dados, o comportamento de uma ou mais variáveis dependentes, com relação ao crescimento de uma outra variável independente. Quando a evolução é observada ao longo do tempo, chamaremos esse conjunto de série temporal.

Vários conjuntos de dados de interesse apresentam essas características, como funções em geral e séries de dados históricas. Séries históricas de valores do PIB, número de internações diárias em um hospital, hits diários em um site de e-commerce ou a evolução das emissões de CO2 ao longo do tempo, para citar alguns exemplos.

Gráficos de linha, como os que empregamos até aqui, são comumente usados para observar a evolução dos dados.

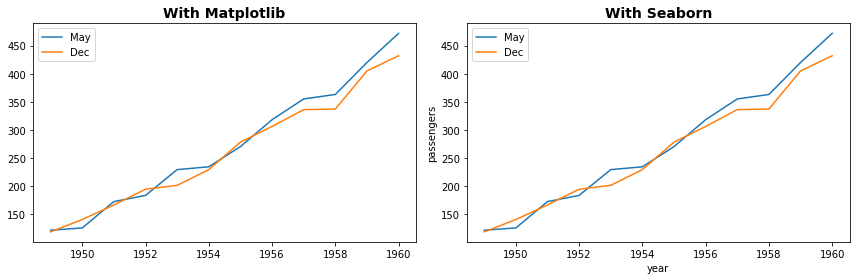
## Gráficos de Linha

Gráficos de linha são úteis para exibir funções e analisar a evolução de dados com relação a uma variável exibindo também a tendência dos dados.

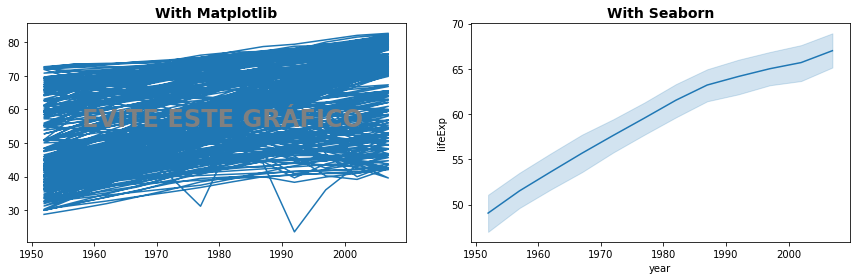
flights = sns.load\_dataset("flights")  
flights.head()

year month passengers  
0 1949 Jan 112  
1 1949 Feb 118  
2 1949 Mar 132  
3 1949 Apr 129  
4 1949 May 121

plt.figure(figsize=(12,4))  
flights\_may = flights[ flights.month == 'May' ]  
flights\_dec = flights[ flights.month == 'Dec' ]  
  
plt.subplot(1,2,1)  
plt.plot(flights\_may.year, flights\_may.passengers, label='May')  
plt.plot(flights\_dec.year, flights\_dec.passengers, label='Dec')  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.legend()  
  
plt.subplot(1,2,2)  
sns.lineplot(x=flights\_may.year, y=flights\_may.passengers, label='May')  
sns.lineplot(x=flights\_dec.year, y=flights\_dec.passengers, label='Dec')  
plt.title('With Seaborn',fontsize=14,weight='bold')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()



# obtendo os dados  
import plotly.express as px  
df = px.data.gapminder()   
  
plt.figure(figsize=(12,4))  
  
plt.subplot(1,2,1)  
plt.plot(df.year, df.lifeExp)  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.text(1958, 55, 'EVITE ESTE GRÁFICO', fontsize=24, c='grey', weight='bold')  
  
plt.subplot(1,2,2)  
sns.lineplot(x=df.year, y=df.lifeExp)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.tight\_layout()  
plt.show()



# Distribuição

Como vimos antes uma das perguntas mais importantes que podemos fazer sobre os dados é sobre sua distribuição e é um tema central da análise univariada. Você por exemplo já ouviu falar de uma distribuição normal. Várias medidas e fenômenos naturais seguem essa distribuição como medidas do corpo humano, como altura, peso, o peso de recém-nascidos, o peso ou tamanho de sementes etc. Conhecer essa distribuição (basicamente sua média e variância) permite tomarmos decisões inferindo valores sobre amostras dos dados. Por exemplo, você pode definir a quantidades camisas que serão produzidas de cada tamanho baseada na distribuição da população, ou ajudar a inferir sobre o tamanho mais adequado de embalagens para receberem frutas sortidas de diferentes tamanhos. Mas existem muitos outros tipos de distribuição e não é por acaso que você já ouviu falar de que na origem de muitos dos problemas que enfrentamos está a distribuição de renda do Brasil. De fato, a renda no Brasil não segue uma distribuição normal e há muito mais brasileiros na faixa de valores de baixa renda que para faixas maiores. A distribuição neste caso é uma distribuição em calda, e é um dos principais fatores de desigualdade no país. Conhecer essa distribuição ajuda a estabelecer políticas públicas como valores de impostos a serem pagos, benefícios e isenções por faixa de renda, assim como conhecer a distribuição das idades da população por sexo permite estabelecer regras previdenciárias mais justas.

## Histogramas

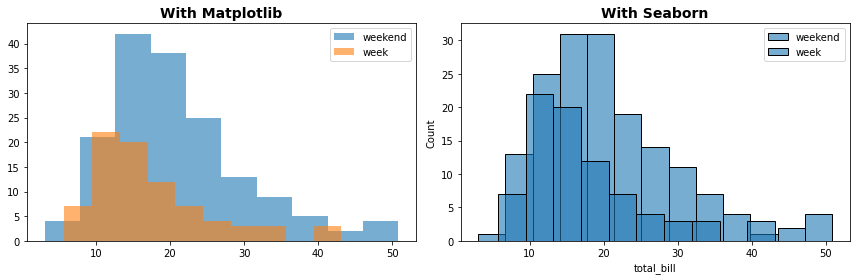
Histogramas são gráficos que exibem a frequência (quantidade) de casos para os valores, ou faixa de valores, de uma variável.

Note que em um histograma, assim como o gráfico de densidade a seguir, você informa **uma única variável**, hist(x), que é exibida em duas de suas dimensões, sua **faixa de valores frequência**, o que difere de um plot(x, y) em que você explicitamente informa as duas dimensões a serem exibidas.

tips = sns.load\_dataset("tips")  
tips.head()

total\_bill tip sex smoker day time size  
0 16.99 1.01 Female No Sun Dinner 2  
1 10.34 1.66 Male No Sun Dinner 3  
2 21.01 3.50 Male No Sun Dinner 3  
3 23.68 3.31 Male No Sun Dinner 2  
4 24.59 3.61 Female No Sun Dinner 4

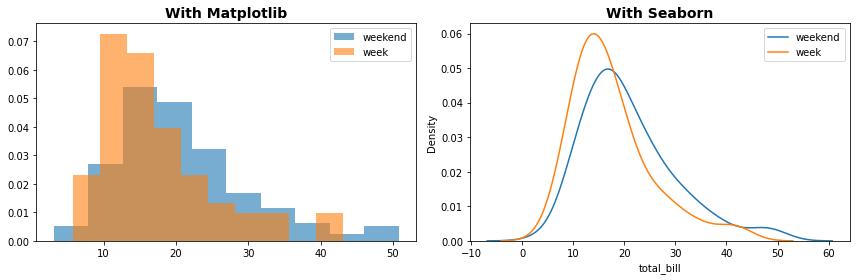
plt.figure(figsize=(12,4))  
tips\_weekend = tips[ tips.day.isin(['Sun','Sat']) ]  
tips\_week = tips[ ~tips.day.isin(['Sun','Sat']) ]  
  
plt.subplot(1,2,1)  
plt.hist(tips\_weekend.total\_bill, alpha=0.6, label='weekend')  
plt.hist(tips\_week.total\_bill, alpha=0.6, label='week')  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.legend()  
  
plt.subplot(1,2,2)  
sns.histplot(tips\_weekend.total\_bill, alpha=0.6, label='weekend')  
sns.histplot(tips\_week.total\_bill, alpha=0.6, label='week')  
plt.title('With Seaborn',fontsize=14,weight='bold')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()



## Gráficos de Densidade

Em princípio gráficos de densidade e histogramas exibem ambos distribuições dos dados. Mas o primeiro apresenta a frequência (quantidade absoluta) para as respectivas faixas de valores. O segundo apresenta a probabilidade dos valores, reescalando as quantidades de forma que a área abaixo do gráfico tenha valor 1 e representando, assim, a razão ou a densidade de probabilidade dos valores.

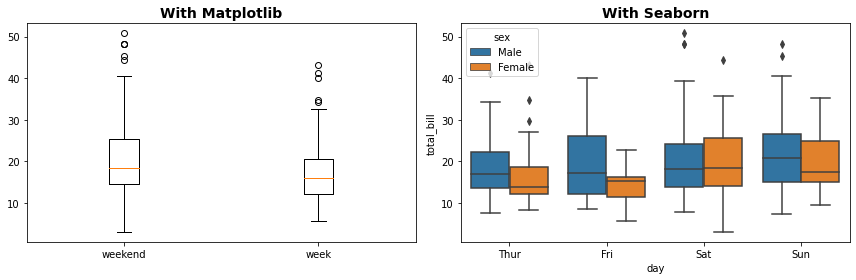
plt.figure(figsize=(12,4))  
tips\_weekend = tips[ tips.day.isin(['Sun','Sat']) ]  
tips\_week = tips[ ~tips.day.isin(['Sun','Sat']) ]  
  
plt.subplot(1,2,1)  
plt.hist(tips\_weekend.total\_bill, density=True, alpha=0.6, label='weekend')  
plt.hist(tips\_week.total\_bill, density=True, alpha=0.6, label='week')  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.legend()  
  
plt.subplot(1,2,2)  
sns.kdeplot(tips\_weekend.total\_bill, alpha=0.6, label='weekend')  
sns.kdeplot(tips\_week.total\_bill, alpha=0.6, label='week')  
plt.title('With Seaborn',fontsize=14,weight='bold')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()



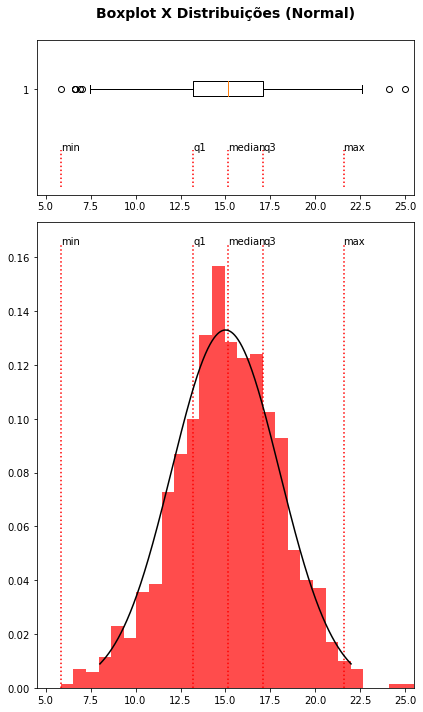
## Gráficos de Caixa ou Boxplot

Gráficos de Caixa ou Boxplot foram introduzidos nos anos 70 por John Tukey e são uma forma sintética de representar distribuições dos dados a partir unicamente dos quartis dos valores. Os quartis, você deve lembrar, são os valores dos dados ordenados em ordem crescente, que dividem a distribuição em quatro partes iguais , correspondendo assim aos valores (mínimo, Q1, mediana, Q3,máximo). Esses valores fornecem uma representação útil, embora bastante resumida, das distribuições dos dados.

plt.figure(figsize=(12,4))  
tips\_weekend = tips[ tips.day.isin(['Sun','Sat']) ]  
tips\_week = tips[ ~tips.day.isin(['Sun','Sat']) ]  
  
plt.subplot(1,2,1)  
plt.boxplot(tips\_weekend.total\_bill, positions=[0])  
plt.boxplot(tips\_week.total\_bill, positions=[1])  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.xticks(ticks=[0,1],labels=['weekend','week'])  
  
plt.subplot(1,2,2)  
sns.boxplot(x='day', y='total\_bill', hue='sex', data=tips)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.tight\_layout()  
plt.show()



O gráfico abaixo permite entender melhor as relações entre um gráfico de caixa e um gráfico de distribuição dos dados.



# Quantidades e Proporções

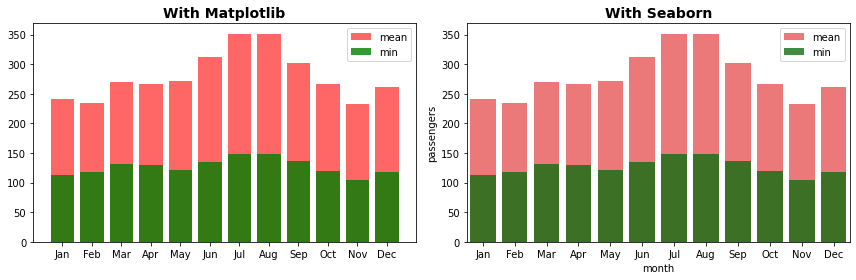
Em muitos casos não estamos interessados na evolução dos dados ou mesmo na sua distribuição, mas queremos acessar os valores diretamente em suas unidades, exibindo esses valores para diferentes classes ou suas proporções (razão entre valores).

Para isso gráficos de barras e suas variantes são em geral os gráficos mais úteis e empregados. Um gráfico de barras apresenta a relação entre uma variável numérica e uma variável categórica sendo cada valor da variável categórica representado por uma barra vertical ou horizontal. O tamanho da barra representa o valor ou as quantidades.

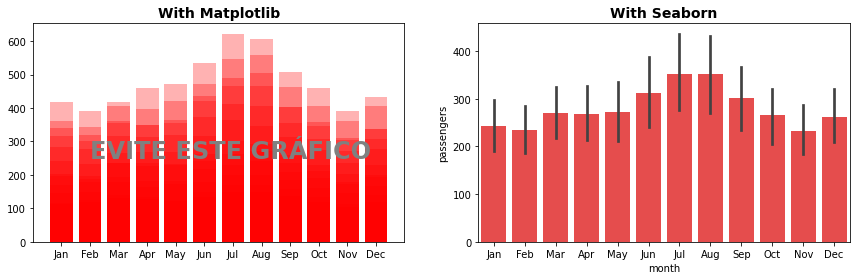
## Gráficos de Barras

Esses gráficos são a ferramenta ideal se você quer exibir e comparar valores ou proporções de diferentes classes de dados.

plt.figure(figsize=(12,4))  
flights\_by\_month = pd.DataFrame( flights.groupby('month').passengers.mean() ).reset\_index()  
flights\_by\_min = pd.DataFrame( flights.groupby('month').passengers.min() ).reset\_index()  
  
plt.subplot(1,2,1)  
plt.bar(flights\_by\_month.month, flights\_by\_month.passengers, color='r', alpha=0.6, label='mean')  
plt.bar(flights\_by\_min.month, flights\_by\_min.passengers, color='g', alpha=0.8, label='min')  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.legend()  
  
plt.subplot(1,2,2)  
sns.barplot(x=flights\_by\_month.month, y=flights\_by\_month.passengers, color='r', alpha=0.6, label='mean')  
sns.barplot(x=flights\_by\_min.month, y=flights\_by\_min.passengers, color='g', alpha=0.8, label='min')  
plt.title('With Seaborn',fontsize=14,weight='bold')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()



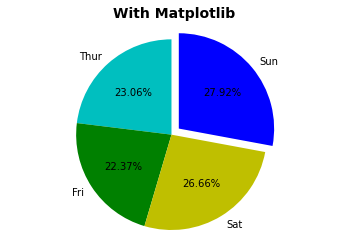
plt.figure(figsize=(12,4))  
  
plt.subplot(1,2,1)  
plt.bar(flights.month, flights.passengers, color='r', alpha=0.3)  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.text(1, 250, 'EVITE ESTE GRÁFICO', fontsize=24, c='grey', weight='bold')  
  
plt.subplot(1,2,2)  
sns.barplot(x=flights.month, y=flights.passengers, color='r', alpha=0.8)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.tight\_layout()  
plt.show()



### Pie Charts

Gráficos de pizza são representações populares, sobretudo na mídia, e são freqüentemente empregados para mostrar as proporções de um todo. Mas são gráficos que encontram pouca aceitação por estatísticos, publicações acadêmicas e científicas, ou outros meios especializados sendo considerados gráficos bastante pobres. Em geral, você deve evitar esses gráficos e você pode ler mais sobre isso nas referências.

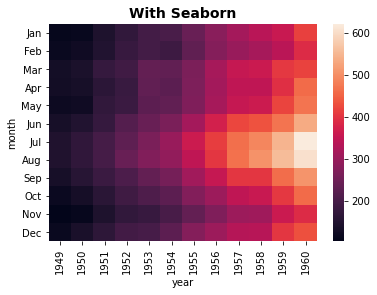
tips\_group = pd.DataFrame( tips.groupby('day').total\_bill.mean() ).reset\_index()  
  
labels = tips\_group.day  
sections = tips\_group.total\_bill   
colors = ['c', 'g', 'y', 'b']  
  
plt.pie(sections, labels=labels, colors=colors,  
 startangle=90,  
 explode = (0, 0, 0, 0.1),  
 autopct = '%1.2f%%')  
  
plt.axis('equal')   
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.show()



## Heat Maps

Mapas de Calor ou Heat Maps permitem visualizar dados multivariados exibindo variáveis ​​discretas (categóricas ou numéricas), nas linhas e colunas, e uma dimensão quantitativa colorindo as células do conjunto retangular de linhas e colunas. Os mapas de calor são ótimas representações para mostrar variações de valores em várias variáveis e identificar grupos de dados que apresentam um mesmo padrão de valores.

flights = flights.pivot("month", "year", "passengers")  
sns.heatmap(flights)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.show()



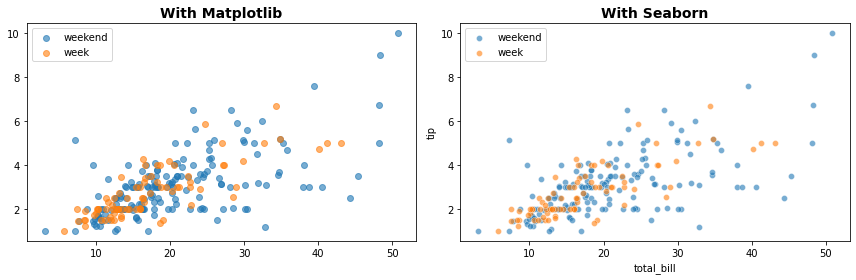
# Relações

Muitos conjuntos de dados contêm duas ou mais variáveis, e podemos estar interessados ​​em como essas variáveis ​​se relacionam entre si. De certo modo muitos dos gráficos podem ajudar nessa análise.

## Gráficos de Dispersão

Os gráficos de dispersão (scatter plots), também chamados de gráficos de pontos ou diagramas de dispersão, são talvez o tipo mais empregado para análise de relações entre duas variáveis quantitativas.

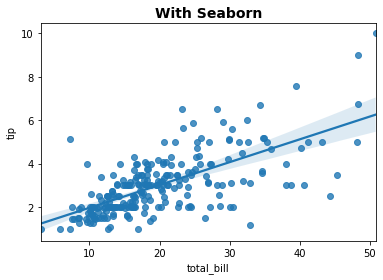
plt.figure(figsize=(12,4))  
tips\_weekend = tips[ tips.day.isin(['Sun','Sat']) ]  
tips\_week = tips[ ~tips.day.isin(['Sun','Sat']) ]  
  
plt.subplot(1,2,1)  
plt.scatter(tips\_weekend.total\_bill, tips\_weekend.tip, alpha=0.6, label='weekend')  
plt.scatter(tips\_week.total\_bill, tips\_week.tip, alpha=0.6, label='week')  
plt.title('With Matplotlib',fontsize=14,weight='bold')  
plt.legend()  
  
plt.subplot(1,2,2)  
sns.scatterplot(x=tips\_weekend.total\_bill, y=tips\_weekend.tip, alpha=0.6, label='weekend')  
sns.scatterplot(x=tips\_week.total\_bill, y=tips\_week.tip, alpha=0.6, label='week')  
plt.title('With Seaborn',fontsize=14,weight='bold')  
plt.legend()  
  
plt.tight\_layout()  
plt.show()



### Adicionando uma linha de tendência

Na análise dessas relações é comum adicionarmos aos gráficos de dispersão linhas de tendências dos dados que nos ajudam a compreender melhor a sua evolução. Isso tem muitas formas de ser feito e aqui vamos nos limitar a adição de uma linha de tendência linear.

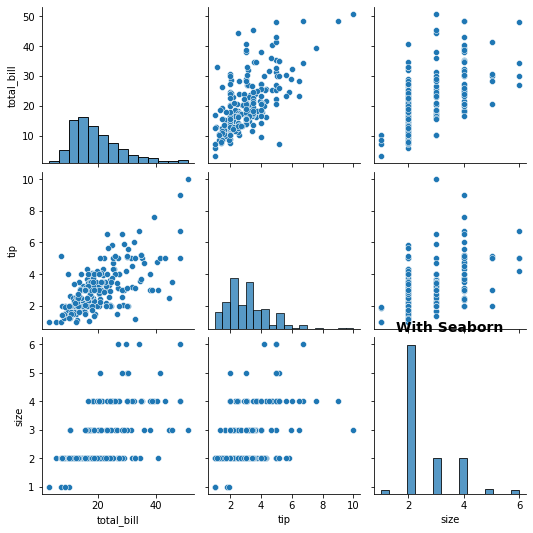
sns.regplot(x='total\_bill', y='tip', data=tips)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.show()



## Correlograma

Correlogramas são simplesmente gráficos de dispersão de diferentes pares de variáveis, e vão permitir separar uma visualização de dados com muitas relações e que são mais fáceis de analisar individualmente. Isso permite evitarmos um problema comum dos gráficos de dispersão que é o *overplotting*, em que exibimos muitas relações simultaneamente.

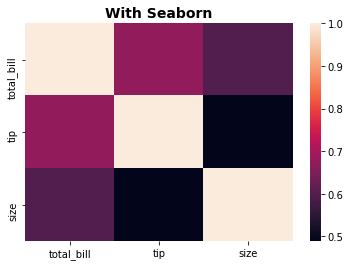
sns.pairplot(data=tips)  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.show()



## Heat Maps, *again*

Mapas de Calor também podem ser empregados para explorar **relações lineares** entre os dados exibindo por diferentes cores os níveis de correlação dos pares de variáveis.

sns.heatmap(tips[['total\_bill','tip','size']].corr())  
plt.title('With Seaborn',fontsize=14,weight='bold')  
  
plt.show()



O gráfico acima permite, do mesmo modo que o correlograma, identificar uma relação mais linear entre *total\_bill* e *tip*, que entre as variáveis *total\_bill* e *size*, ou size e tip. Mas a correlação só permite explorar relações lineares entre os dados e o correlograma é mais geral, podendo exibir outros padrões dos dados.

# Outros Gráficos

Como uma introdução, esse texto não aborda nem todas as formas de visualização de dados e nem muitas outras bibliotecas que podem ser muito úteis na produção de gráficos mais especializados como nuvens de palavras (*word clouds*), gráficos de mapas, diagramas de calor, construção de *dashboards*, gráficos 3D etc. No ambiente Python a maior parte delas, entretanto, é construída sobre o pacote Matplotlib, assim como o Seaborn e os conceitos aqui aprendidos serão bastante úteis para você empregá-las. O próprio Pandas implementa uma integração útil com o Matplotlib para a visualização de DataFrames.

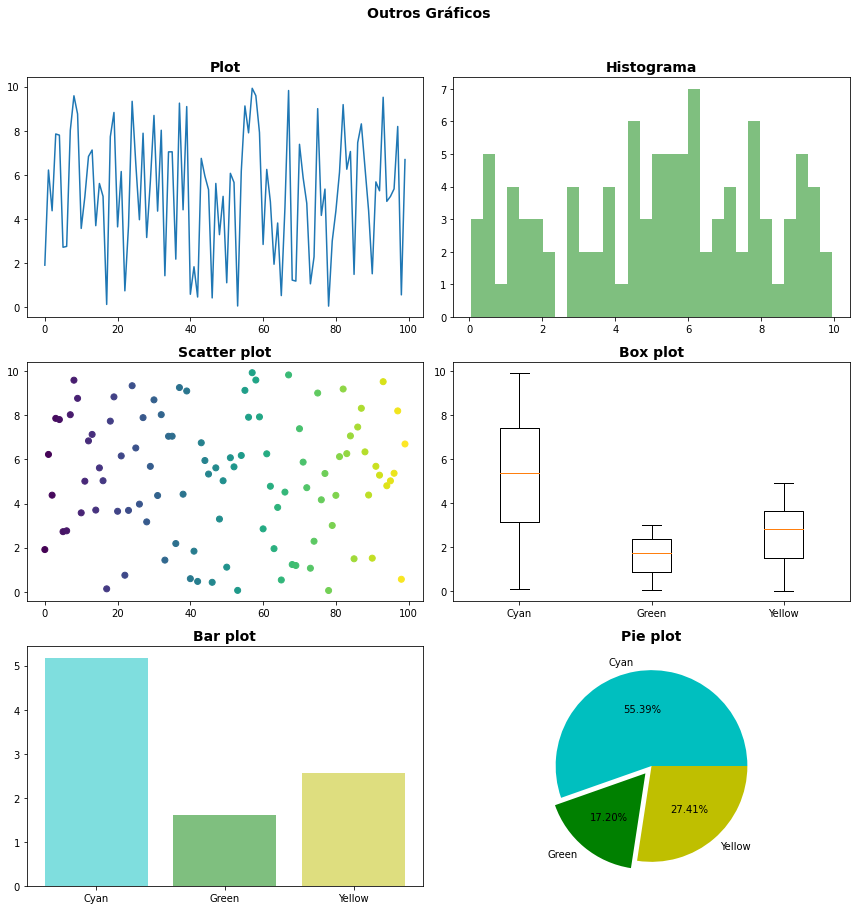
Fica como sugestão você explorar o pacote plotly (exemplo abaixo) que permite produzir várias das visualizações que criamos aqui em gráficos *dinâmicos* e a integração do Pandas com o Matplotlib.

import plotly.express as px  
df = px.data.gapminder() # .query("country=='Brazil'")  
df = df[ df.country.isin(['Brazil','France','United States','Angola','Japan','China','Somalia','India'])]  
fig = px.line(df, x="year", y="lifeExp", color= "country" ,title='Expectativa de Vida dos Países')  
fig.show()

# Resumo Matplotib

O código abaixo resume, de modo bastante simples, os principais gráficos do Matplotlib.

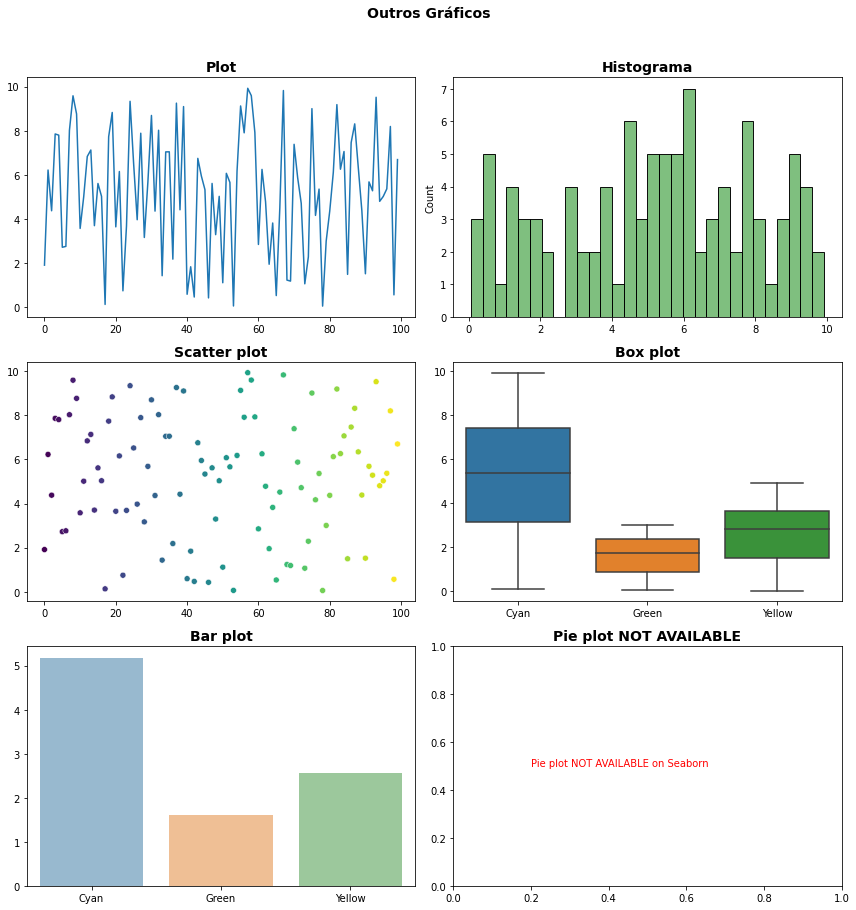
# apenas a produção dos dados  
x = np.random.random\_sample(100)  
np.random.seed(1234)  
  
x = np.arange(0,100,1)  
y = np.random.rand(100)\*10  
z = np.random.rand(100)\*3  
t = np.random.rand(100)\*5  
  
from matplotlib import cm # para cores no scatter plot  
  
plt.figure(figsize=(12,12))   
plt.suptitle('Outros Gráficos', fontsize=14, weight='bold', y=1.05)  
  
plt.subplot(3, 2, 1)   
plt.plot(x,y)   
plt.title('Plot', fontsize=14, weight='bold')   
  
plt.subplot(3, 2, 2)   
plt.hist(y, bins=30, color='g', alpha=0.5)   
plt.title('Histograma', fontsize=14, weight='bold')   
   
plt.subplot(3, 2, 3)   
plt.scatter(x,y,c=x,cmap=cm.viridis)   
plt.title('Scatter plot', fontsize=14, weight='bold')   
  
plt.subplot(3, 2, 4)   
plt.boxplot([y,z,t],labels=['Cyan','Green','Yellow'])   
plt.title('Box plot', fontsize=14, weight='bold')  
  
plt.subplot(3, 2, 5)   
plt.bar(['Cyan','Green','Yellow'],[y.mean(),z.mean(),t.mean()],color=['c','g','y'],alpha=0.5)   
plt.title('Bar plot', fontsize=14, weight='bold')  
  
plt.subplot(3, 2, 6)   
plt.pie([y.mean(),z.mean(),t.mean()],  
 colors=['c','g','y'],  
 explode=[0, 0.1, 0],  
 labels=['Cyan','Green','Yellow'],  
 autopct='%1.2f%%')   
plt.title('Pie plot', fontsize=14, weight='bold')  
  
plt.tight\_layout()  
plt.show()



# Resumo Seaborn

O código abaixo traz os mesmos gráficos produzidos anteriormente como o Matplotlib com o Seaborn, sendo um resumo, também de modo bastante simples, os principais gráficos do Seaborn.

import seaborn as sns  
import pandas as pd  
  
# apenas a produção dos dados  
x = np.random.random\_sample(100)  
np.random.seed(1234)  
  
x = np.arange(0,100,1)  
y = np.random.rand(100)\*10  
z = np.random.rand(100)\*3  
t = np.random.rand(100)\*5  
  
from matplotlib import cm # para cores no scatter plot  
  
fig, ax = plt.subplots(3,2,figsize=(12,12))   
   
plt.suptitle('Outros Gráficos', fontsize=14, weight='bold', y=1.05)  
  
sns.lineplot(x=x,y=y,ax=ax[0,0]) # x= e y= para evitar warnings....  
ax[0,0].set\_title('Plot', fontsize=14, weight='bold')   
  
sns.histplot(y, bins=30, color='g', alpha=0.5,ax=ax[0,1]) # para subplots somente aceite facegrid, um recurso avançado para múltiplos plots  
ax[0,1].set\_title('Histograma', fontsize=14, weight='bold')  
   
sns.scatterplot(x=x,y=y,c=x,cmap=cm.viridis,ax=ax[1,0]) # x= e y= para evitar warnings....  
ax[1,0].set\_title('Scatter plot', fontsize=14, weight='bold')   
  
box = pd.DataFrame({'Cyan':y,'Green':z,'Yellow':t}) # boxplot para ser simples empregamos um dataframe para uso da função melt   
sns.boxplot(data=box,ax=ax[1,1])   
ax[1,1].set\_title('Box plot', fontsize=14, weight='bold')  
  
sns.barplot(x=['Cyan','Green','Yellow'],y=[y.mean(),z.mean(),t.mean()],alpha=0.5,ax=ax[2,0]) # x= e y= para evitar warnings....   
ax[2,0].set\_title('Bar plot', fontsize=14, weight='bold')  
  
# não temos pie plot no seaborn  
plt.text(0.2, 0.5,'Pie plot NOT AVAILABLE on Seaborn', color='r')  
ax[2,1].set\_title('Pie plot NOT AVAILABLE', fontsize=14, weight='bold')  
  
plt.tight\_layout()  
plt.show()

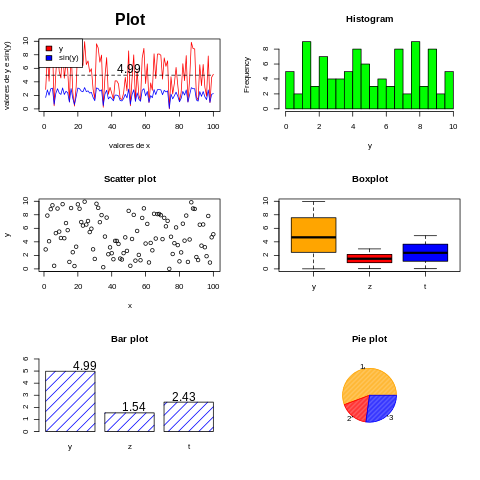


# Resumo R

Os mesmos gráficos podem ser produzidos em R. Como estamos em um notebook Python empregamos %%R para indicar a execução do kernel de R na célula depois de carregar o seu runtime.

%load\_ext rpy2.ipython

%%R  
  
set.seed(123)  
  
x = seq(1,100,1)  
y = runif(100)\*10  
z = runif(100)\*3  
t = runif(100)\*5  
  
par(mfrow = c(3, 2)) # múltiplos gráficos na mesma página  
  
plot(x, y, type='l', main='Plot', col='red', xlab = 'valores de x', ylab ='valores de y e sin(y)', cex.main=2)  
# cex.lab for axis labels and cex.axis for axis text   
lines(x, sqrt(y), type='l', col='blue')  
legend("topleft",c("y","sin(y)"),fill=c("red","blue"))  
lines(c(1,100), c(mean(y),mean(y)), type='l', col='black', lty=2)  
text(50, mean(y)+1, round(mean(y),2), cex=1.5)  
  
hist(y, breaks = 30, main='Histogram' , col='green' )  
  
plot(x, y, main='Scatter plot')  
  
boxplot(y,z,t,names = c("y", "z", "t"), col = c("orange","red","blue"), main='Boxplot' )   
  
barplot(c(mean(y), mean(z), mean(t)),names = c("y", "z", "t"), col='blue',density=10 , main='Bar plot', ylim=c(0,6))  
text(c(1,2,3), c(mean(y)+0.5, mean(z)+0.5, mean(t)+0.5), labels=c(round(mean(y),2), round(mean(z),2), round(mean(t),2)), cex=1.5)  
  
pie(c(mean(y), mean(z), mean(t)), col=c("orange","red","blue") ,density=80, main='Pie plot' )



Adicionamos aqui alguns ornamentos que não tínhamos feito antes em R para que você possa empregá-los se quiser (textos, labels, cores, tamanhos de fonte etc.).

O Matplotlib e o Seaborn são muito mais poderosos que as funções básicas de plot oferecidas pelo R. Mas você deve lembrar que o R ainda possui o pacote ggplot2. Este pacote tem muito mais capacidade para visualização de dados e é hoje a ferramenta mais profissional para produzir gráficos a partir de programas e, no momento, é até mais potente que as bibliotecas de Python.

# Para Saber Mais

1. Veja aqui uma palestra recente de Wes McKinney, o desenvolvedor inicial do Pandas, **Wes McKinney - Keynote - PyCon Colombia 2020**, em: <https://youtu.be/ZTXFQ2sEarQ>
2. Revise aqui o essencial de Pandas em **Pandas Cheat Sheet for Data Science in Python**, em: <https://www.datacamp.com/community/blog/python-pandas-cheat-sheet> um rápido tutorial de Karlijn Willems ou ainda pode acessar o Tutorial do Pandas, <https://pandas.pydata.org/pandas-docs/stable/getting_started/index.html>
3. Acesse aqui Selva Prabhakaran. **Top 50 matplotlib Visualizations – The Master Plots (with full python code)** <https://www.machinelearningplus.com/plots/top-50-matplotlib-visualizations-the-master-plots-python/> E veja inúmeros exemplos de gráficos com o Matplotlib.
4. Se estiver interessado em se aprofundar em visualizações de dados com Python você encontra um texto bastante acessível em: Rogério de Oliveira. **Introdução à Visualização de Dados com Python**. Ed. Mackenzie. Coleção Conexão Inicial. *pre-print (2021)* Disponível em: <https://github.com/Rogerio-mack/Visualizacao-de-Dados-em-Python>

# Referências

1. \_\_\_. **Scipy Lecture Notes** Disponível em: <http://scipy-lectures.org/intro/matplotlib/index.html> Acesso: 12.09.2021. Alternativamente em: <https://scipy-lectures.org/_downloads/ScipyLectures-simple.pdf>
2. Claus O. Wilke. **Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures** (2019) Disponível em: <https://clauswilke.com/dataviz/> Acesso: 17.05.2021
3. Jake VanderPlas. **Python Data Science Handbook**. O'Reilly Media, Inc. (2016). ISBN: 9781491912058. Disponível em: <https://jakevdp.github.io/PythonDataScienceHandbook/>. Acesso: 17.05.2021
4. Rogério de Oliveira. **Introdução à Visualização de Dados com Python**. Ed. Mackenzie. Coleção Conexão Inicial. *pre-print (2021)* Disponível em: <https://github.com/Rogerio-mack/Visualizacao-de-Dados-em-Python.Acesso>: 17.05.2021
5. Phuong, Vo. T.H.; Czygan, Martin. **Getting Started with Python Data Analysis**. Birmingham: Packt Publishing, 2015.
6. McKinney Wes. **Python for data analysis (1st. ed.)**. O'Reilly Media, Inc. (2012).
7. \_\_\_. **Pandas**. <https://pandas.pydata.org/>