Introdução ao Python para Tratamento de Dados

Hugo Everaldo Salvador Bezerra

11 de março de 2025

Sumário

1 Introdução

Este material é composto de notas de aulas ministradas para pessoas que conhecem teoria de algoritmos ou já tem experiência com outra linguagem de programação, mas tem pouca ou nenhuma experiência com Python. O objetivo é fazer uma introdução ao Python e citar referências importantes na área de computação científica e tratamento de dados.

Um dos pontos fortes do material são links das funções e pacotes citados para acesso à tutoriais, páginas oficiais dos pacotes, textos complementares e manuais. A maioria das funções citadas tem link para acesso a texto de suporte para utilização da função.

Para geração do PDF foi utilizado o Colab, realizado tratamento em LaTeX e então gerado o arquivo PDF.

[1]:
```
# Versão do Python utilizada neste material
import sys
sys.version
```

[1]:
```
'3.12.7 | packaged by Anaconda, Inc. | (main, Oct  4 2024, 13:17:27) [MSC v.1929
64 bit (AMD64)]'
```

2 Bibliografia sugerida

Listamos algumas referências de material de apoio para aprofundamento nos temas abordados.

- Documentação Oficial do Python: documentação completa sobre a linguagem com tutoriais e referências em várias versões do Python.
- Pense em Python: Livro em português disponibilizado online gratuitamente.
- Python Fluente: Segunda edição de livro disponibilizada gratuitamente no site com detalhes sobre Python. Escrito pelo renomado autor da área, Luciano Ramalho, teve a primeira edição do livro publicado em nove idiomas.

- Guide to NymPy: Livro escrito por Travis E. Oliphant, criador do NumPy, escrito em 2006 que é disponibilizado gratuitamente.
- NumPy User Guide (web/pdf): Guia on-line e em arquivo PDF, atualizado pela equipe que mantém o pacote.
- NumPy Reference: Referência disponibilizada on-line pela equipe que mantém o pacote.
- Pandas Documentation(web/pdf): Guia disponibilizado on-line e em arquivo PDF pela equipe que mantém o pacote.
- Python for Data Analys: terceira edição lançada em 2022 do livro escrito pelo criador do pandas, Wes Mckinney. Versão aberta disponibilizada on-line.

3 Introdução ao Python

3.1 Contexto

3.1.1 Características da linguagem

- Licença Open Source (pode ser reproduzida, analisada, testada, executar e/ou exibida publicamente, preparado trabalhos derivados, distribuída e usada)
- Fácil de iniciar programando
- Inglês das linguagens de programação, transpassa por várias áreas como computação científica, Inteligência Artificial, desenvolvimento web, manutenção de máquina e **Automação Robótica de Processos** - RPA (do inglês *Robotic Process Automation*) entre outras.
- Linguagem de alto nível, onde não é necessário se preocupar ao escrever o código com detalhes como gerenciamento de utilização de memória da máquina.
- Linguagem interpretada, não havendo necessidade de compilar o código antes de executar. Esta característica ajuda bastante em testar previamente função que será utilizada. Podemos rodar apenas a linha que estamos estudando para entender sua utilização e testá-la antes de inserir no código.
- Multiparadigma: fortemente Orientado a Objetos, Procedural e Funcional.
- Tipagem dinâmica de variável, diferente de linguagens como C/C++ ou Java onde as variáveis devem ser declaradas antes de sua utilização.
- Possui um vasto repertório de bibliotecas (mais de 6000 mil pacotes listados do repositório *Python Package Index* - Pypi em março de 2025).
- Expansível com C/C++ ou Fortran, melhorando o desempenho e performance.
- Extremamente portável, podendo rodar o código em sistemas operacionais como Unix, GNU/Linux, Windows, Mac ou embarcado em microcontroladores.

3.1.2 Usando Python

Por ser uma linguagem interpretada, o Python pode ser usada com comando interativos em um prompt de comando padrão da linguagem. Uma boa alternativa é o IPython que conta com autocompletar, histórico de comandos e gráficos integrados, ideal para análise de dados.

É possível escrever código Python em qualquer editor de texto (ex. Bloco de notas do Windows, Notepad++, Visual Studio Code da Microsoft, Vim), mas existem Ambientes Integrados de Desenvolvimento ou IDEs (*Integrated Development Environment*) específicos como o Spyder e Jupyter. O Spyder é uma IDE que parece familiar para pessoas que estão acostumadas com IDEs do Matlab e

RStudio. Jupyter é um IDE que roda diretamente em um navegador de internet e é muito utilizado para tratamento de dados. Atualmente o Visual Studio Code (VS Code) está sendo muito utilizado para edição de códigos Python com ajuda de extensões que facilitam e agilizam a codificação. Também é possívek utilizar extensões como GitHub Copilot que é uma ferramenta de auxílio de escrita de código que usa Inteligência Artificial para sugerir código enquanto se digita ou usando o bate-papo no editor para escrever código mais rapidamente.

O Colab é um IDE disponibilizado on-line para qualquer um que tenha uma conta no Google e é muito semelhante ao Jupyter. É uma boa alternativa para quem quer iniciar aprendendo Python sem instalar qualquer programa na máquina local. Além disso, o ambiente já conta com várias bibliotecas já instaladas como NumPy e Matplotilb. Tanto o Colab como o Jupyter usam o IPython como núcleo.

No caso de se optar por instalar o Python em uma máquina local, as melhores opções são a utilização do Anaconda que instala não só o Python como também vários pacotes incluindo o `NumPy`, `pandas`, `matplotlib` e `seaborn` que serão abordos neste material) e ferramentas adicionais como o Jupyter e o Spyder. No repositório do Anaconda existem várias versões disponíveis. Se a intenção não é instalar uma solução completa como o Anaconda, uma boa alternativa é o Miniconda que é uma instalação mais enxuta, mas já vem com funcionalidades importantes como gerenciador de pacotes melhor que o original do Python e controle de ambientes isolados.

3.2 Variáveis

O Python trabalha com uma grande variedade de Modelos de Dados, mas nesta seção serão considerados os tipos básicos para texto, numérico e sequência.

Tipos básicas	Descrição	Exemplo
strings	Texto	‘spam’, “Bob’s”, “Python é massa”"“,”1234"
int	Número Inteiro	1234
float	Número Real	3.14159
complex	Número Complexo	3+4j
bool	Lógico	True, False
lists	Lista	[1, [2, ‘three’], 4.5], [1,2,3,4], [‘casa’, ‘carro’, ‘bola’]
dict	Dicionário	{‘food’: ‘spam’, ‘taste’: ‘yum’}, {‘nome’: ‘João’, ‘idade’: 32}
tuples	Tupla	(1, ‘spam’, 4, ‘U’), (1,2,3)
set	Conjunto	{‘r’, ‘g’, ‘b’}, {10, 20, 40}

3.2.1 Números

O Python trabalha com 4 tipos básicos de valores numéricos: inteiros (`int`), números reais (`float`), números complexos (`complex`) e booleanos (`bool`). A precisão e intervalo de armazenamento em cada tipo varia de acordo com a arquitetura da máquina onde o script Python está sendo executado.

Vamos passar de forma rápida os principais operadores no Python utilizados com números. Note que o texto após o símbolo `#` é considerado um comentário, ou seja, o Python não tenta interpretar

e executar este texto que serve apenas para ajudar as pessoas que escrevem e leem os códigos documenta-lo.

[2]:
```
1.222 + 5.32  # Soma
```

[2]:
```
6.542
```

[3]:
```
3 * 5.5  # Multiplicação
```

[3]:
```
16.5
```

[4]:
```
7/3  # Divisão
```

[4]:
```
2.3333333333333335
```

[5]:
```
7 // 3 # Divisão com resultado inteiro
```

[5]:
```
2
```

[6]:
```
7 % 3  # Resto da divisão
```

[6]:
```
1
```

[7]:
```
2 ** 4 # Potência
```

[7]:
```
16
```

O Python é uma linguagem de propósito geral, diferente de linguagens como MATLAB, Octave e Julia voltadas para computação científica ou R criada para análise de dados e estatística. Muitas funções matemáticas básicas já são carregadas na memória e disponibilizadas para utilização ao iniciar o ambiente destas linguagens, por vezes disponibilizando até base de dados para testes.

Por ser uma linguagem de propósito geral, o Python necessita que sejam utilizados pacotes que tem funções específicas para cada aplicação. Existem pacotes voltados para desenvolvimento Web, análise e modelagem estatística, inteligência artificial, finanças, automatização de tarefas, desenvolvimento de jogos, bioinformática, entre outras. O pacotes math faz parte do conjunto de pacotes básico na maioria das instalações do Python, outros como o NumPy e SciPy usualmente necessitam de instalação após a instalação do Python. Instaladores como o Anaconda já trazem pacotes necessários à computação científica.

[8]:
```
import math
from math import pi, cos

print('pi = ', math.pi)

print('sen(π/2) = ', math.sin(math.pi/2))

print('cos(π/2) = ', cos(pi))
```

```
print('Tipo da variável math.pi: ',type(pi))
```

```
pi =  3.141592653589793
sen(π/2) =  1.0
cos(π/2) =  -1.0
Tipo da variável math.pi:  <class 'float'>
```

Acima, importamos o pacote `math` onde estão as funções básicas de matemática em Python. Os pacotes são uma forma de agrupar funções e variáveis para um fim específico. Os pacotes não são carregados por padrão pelo Python para evitar que existem muitas funções carregadas na memória que não serão utilizadas.

Existem três formas de importar pacotes no Python:

```
import <pacote>                 # utilização: <pacote>.<função>
from <pacote> import <função>   # utilização: <função>
from <pacote> import *          # utilização: <função>
```

Quando importamos um pacote utilizando a palavra `import` seguido o nome do pacote sempre precisamos definir o pacote e a função que queremos usar. No exemplo acima, importamos o pacote `math` utilizando `import` e para utilizar a função `sin` foi necessário a utilização na forma `math.sin`.

Na linha seguinte usamos `from math import pi, cos`. Ao utilizar a função não foi necessário definir o nome do pacote de onde a função foi importada, já que desta forma a função e a variável foram carregadas direto na memória. Da mesma forma poderíamos utilizar apenas a função sem definir o pacote se a importação fosse feita utilizando `from <pacote> import *`. A desvantagem da última forma é que todas as funções existentes no pacote seriam carregadas na memória e se importarmos desta forma mais de um pacote corremos o risco de ter funções com mesmo nome em pacotes distintos, podendo causar confusão para saber de que pacote a função utilizada pertence.

Como opção se pode definir uma *alias*, um apelido para não precisar usa o nome inteiro do pacote:

```
import <pacote> as <alias>  # utilização: <alias>.<função>
```

Neste caso podemos resumir o nome do pacote e ter certeza de que pacote a função que estamos usando pertence.

Daqui para frente teremos vários exemplos de importação e de utilização de funções de pacotes específicos.

[9]:
```
# Exemplo de utilização de número complexo
c1 = 3 + 4j

print(type(c1))  # Mostrar o tipo de variável

c2 = 5 + 8j

print(c1*c2)  # Resultado da multiplicação de dois complexos
```

```
<class 'complex'>
(-17+44j)
```

Existem formas de trabalhar com números em outros formatos como em forma de fração, em sistema binário, hexadecimais e octadecimais.

[10]:
```
# Trabalhando com números em forma de fração
from fractions import Fraction

print(Fraction(1,5) + Fraction(4,10) + 1)
```

```
8/5
```

[11]:
```
# Trabalhando com números binários, hexadecimais e octadecimais

# Print de números no sistema decimal
print(0b101010) # binário
print(0xb0ca)   # hexadecimal
print(0o177)    # octadecimal
print()         # Linha em branco

# Print de números do sistema decimal em binários, hexadecimais ou octadecimais
print(bin(42))
print(hex(15815114))
print(oct(127))
```

```
42
45258
127

0b101010
0xf151ca
0o177
```

Abaixo vemos a versatilidade de trabalhar de forma direta em sistemas decimais, binários, hexadecimais e octadecimais. Estamos somando um número no sistema binário, com um número no sistema decimal, com número no sistema octadecimal e dando o resultado no sistema hexadecimal.

[12]:
```
hex(0b101010 + 100 + 0o177)
```

[12]:
```
'0x10d'
```

Podemos fazer operações matemáticas entre números do tipo `bool`, `int`, `float` e `complex` de forma transparente, sem nos preocuparmos em converter o tipo do número. Se um número `complex` estiver envolvido na operação, o resultado será um número complexo, se um número `float` estiver em uma operação sem um número `complex` presente o resultado será `float` e assim por diante, seguindo a priorização `bool` -> `int` -> `float` -> `complex`.

[13]:
```
# bool -> int -> float -> complex

etcha = True + 2 * 1.1 / 4j
print(etcha)
print(type(etcha))
```

```
(1-0.55j)
<class 'complex'>
```

Existem atalhos para se trabalhar com variáveis que facilitam o desenvolvimento de scripts. Uma que sempre é citado é troca de valores de variáveis utilizando uma linha, que nem toda linguagem de programação consegue realizar.

[14]:
```
a = 3
b = 5

a, b = b, a

print('a: ', a)
print('b: ', b)
```

```
a:  5
b:  3
```

No exemplo a seguir importamos as funções `mean` (média) e `pstdev`(desvio padrão populacional) do pacote `statistics`. Utilizamos como nome da variável μ que não faz parte dos caracteres no padrão ASCII, mas faz parte do padrão Unicode. Isso possibilita que utilizemos em nossos scripts variáveis com letras gregas, letras com acentos e caracteres como ç.

[15]:
```
from statistics import mean, pstdev
# Python aceita Unicode no código para, por exemplo, nomear variáveis
ações = [1,2,3,4,5,6]
μ = mean(ações)
σ = pstdev(ações)
print('Média = ', μ)
print('Desvio padrão = ', σ)
```

```
Média =  3.5
Desvio padrão =  1.707825127659933
```

A partir do Python 3.6 foi adicionada a string formatada, uma forma muito fácil e poderosa de formatar `strings` utilizando uma minilinguagem de especificação de formato. Existem outros meios de formatar string no Python, mas a string formatada é a mais utilizada por desenvolvedores Python. Abaixo temos a definição de como formatar uma `string`, mais no intuito de ser usado como mnemônico. Até se habituar com a utilização da string formatada é útil recorrer a um tutorial ou documentação de referência.

Para indicar que o Python de interpretar como uma string formatada é colocado um `f` imediatamente antes da string.

```
f'{:[[preencher]alinhamento][sinal]["z"]["#"]["0"][tamanho][agrupamento]["."precisão][tipo]}'
```

preencher : <qualquer caracter>
alinhamento : “<” | “>” | “=” | “^”
sinal : “+” | “-” |
tamanho : quantidade de dígitos
agrupamento : "_"| “,”

precisão : quantidade de dígitos
tipo : “b” | “c” | “d” | “e” | “E” | “f” | “F” | “g” | “G” | “n” | “o” | “s” | “x” | “X” | “%”

[16]:
```
a = 0.1298731
b = 35466.3108012
c = 1231

print(f'Números {a+1}, {b} e {c**2} serão mostrados.')
```

```
Números 1.1298731, 35466.3108012 e 1515361 serão mostrados.
```

[17]:
```
print(f'Número a {a:->20.2%} em percentual')
print(f'Número b {b:+<30.2e} expoente')
print(f'Número b {b:*^35,.4f} float')
print(f'Número c {c:"^20_b} em binário')
print(f'Número c {c:^20_b} novamente')
```

```
Número a --------------12.99% em percentual
Número b 3.55e+04++++++++++++++++++++++ expoente
Número b ***********35,466.3108*********** float
Número c """100_1100_1111""" em binário
Número c    100_1100_1111     novamente
```

[18]:
```
print(f'Número a {a*2:->20,.2f} foi multiplicado por 2')
print(f'Número b {b*3:->20,.2f} foi multiplicado por 3')
print(f'Número c {c*4:->20,.2f} foi multiplicado por 4')
```

```
Número a ----------------0.26 foi multiplicado por 2
Número b ----------106,398.93 foi multiplicado por 3
Número c ------------4,924.00 foi multiplicado por 4
```

Para definir a configuração brasileira de numeração (“.” para separação de milhar e “,” para separação decimal) a biblioteca padrão `locale` pode ser utilizada e definida formação com tipo `n`:

[19]:
```
import locale
locale.setlocale(locale.LC_NUMERIC, 'pt_BR')

print(f'Número real {a:n}')
print(f'Número real {b:n}')
print(f'Número inteiro {c:n}')
```

```
Número real 0,129873
Número real 35.466,3
Número inteiro 1.231
```

Abaixo temos um modo fácil e direto de imprimir nome e valor da variável.

[20]:
```
print(f'{a=} e {b=}')
```

```
a=0.1298731 e b=35466.3108012
```

3.2.2 String

Variáveis de texto são declaradas utilizando aspas simples ' ou duplas ". `strings` em Python são por padrão da Classe Unicode, ou seja, aceitam caracteres como letras com acento, ç e caracteres especiais.

Abaixo temos um exemplo associação de uma `string` com a variável `s`.

[21]:
```
# Nova variável s como string
s = '!!!   Python é "massa"! Fácil, versátil e 100% grátis.   !!!'
print(s)
```

```
!!!   Python é "massa"! Fácil, versátil e 100% grátis.   !!!
```

Em Python, tudo é um objeto, incluindo variáveis. Isso significa que quando você cria uma variável, na verdade está criando um objeto que contém um valor específico. Toda variável conta com métodos que podem ser utilizados para modificar ou realizar testes na variável. Isso significa que você pode realizar operações e acessar informações sobre esses objetos através de métodos associados a eles.

Abaixo temos exemplo de métodos que transformam o texto salvo na variável em minúscula, em maiúscula e com as primeiras letras em maiúsculas.

[22]:
```
print(s.lower())        # Texto em minúsculo
print(s.upper())        # Texto em maiúsculo
print(s.title())        # Texto em formato de título
```

```
!!!   python é "massa"! fácil, versátil e 100% grátis.   !!!
!!!   PYTHON É "MASSA"! FÁCIL, VERSÁTIL E 100% GRÁTIS.   !!!
!!!   Python É "Massa"! Fácil, Versátil E 100% Grátis.   !!!
```

Abaixo estão listados os métodos que podem ser utilizados em uma variável do tipo `string`.

capitalize	index	isspace	removesuffix	startswith
casefold	isalnum	istitle	replace	strip
center	isalpha	isupper	rfind	swapcase
count	isascii	join	rindex	title
encode	isdecimal	ljust	rjust	translate
endswith	isdigit	lower	rpartition	upper
expandtabs	isidentifier	lstrip	rsplit	zfill
find	islower	maketrans	rstrip	
format	isnumeric	partition	split	
format_map	isprintable	removeprefix	splitlines	

Mais alguns exemplos de modificação do valor da variável utilizando métodos.

[23]:
```
print(s)
print(s.strip('!'))               # Limpar espaços vazios dos extremos
print(s.rstrip('!'))              # Limpar espaços vazios da direita
print(s.strip('!').strip())       # Limpar espaços e exclamações
s = s.strip('!').strip().upper()  # Limpar espaços, ! e colocar maiúsculo
print(s)
```

```
!!!   Python é "massa"! Fácil, versátil e 100% grátis.   !!!
   Python é "massa"! Fácil, versátil e 100% grátis.
!!!   Python é "massa"! Fácil, versátil e 100% grátis.
Python é "massa"! Fácil, versátil e 100% grátis.
PYTHON É "MASSA"! FÁCIL, VERSÁTIL E 100% GRÁTIS.
```

[24]:
```
print(s.replace('I', 'i'))  # Substituir I por i
print(s.count('R'))         # Contar quantidade de R
print(s.split())            # Separar usando espaço
print(s.split('R'))         # Separar usando R
```

```
PYTHON É "MASSA"! FÁCiL, VERSÁTiL E 100% GRÁTiS.
2
['PYTHON', 'É', '"MASSA"!', 'FÁCIL,', 'VERSÁTIL', 'E', '100%', 'GRÁTIS.']
['PYTHON É "MASSA"! FÁCIL, VE', 'SÁTIL E 100% G', 'ÁTIS.']
```

Uma forma muito poderosa de trabalhar com `strings` oir meio de padrões são as Expressões Regulares. Expressões Regulares é uma metalinguagem de definição de padrões de texto utilizados para lidar com combinações de caracteres em uma `string`. Trata-se de um assunto extenso com livros dedicados ao assunto.

Aqui nos vamos nos ater a como a implantação de Expressões Regulares no Python por meio da biblioteca `re`. Seguem exemplos de como realizar seleção, separação e substituição de parte de texto.

[25]:
```
import re

#  ?  - 0 ou 1 ocorrências
#  *  - 0 ou mais ocorrências
#  +  - 1 ou mais ocorrências
#  \w - caracteres do alfabeto inglês
#  \W - caracteres que não estão no conjunto definido por \w
#  \d - dígitos numéricos
#  \D - caracteres que não estão no conjunto definido por \d
#  \s - caractere espaço simples
#  \S - caracteres que não estão no conjunto definido por \s
#  \t - caractere tab
#  \n - caractere de nova linha
```

[26]:
```
# Encontrar entre duas e 4 ocorrências consecuticas do dígito 5
re.findall(r'5{2,4}', 'adft123555554855759')
```

[26]:
```
['5555', '55']
```

No Python, uma *raw string* (string bruta) é uma string prefixada com a letra `r`, como em `r'texto'`. Esse prefixo instrui o interpretador a tratar barras invertidas () como caracteres literais, sem interpretá-las como sequências de escape. Isso é especialmente útil ao trabalhar com expressões regulares, onde muitas sequências começam com (como `\d`, `\w`, `\s`). Se não utilizarmos uma *raw string*, o Python pode interpretar `\d` como uma sequência de escape inválida, resultando em *warnings* ou erros em versões mais recentes do Python. Por exemplo, ao escrever `re.search(r'\d+', 'teste123')`, garantimos que `\d+` seja passado corretamente para o módulo `re`, sem precisar du-

plicar as barras (`'\\d+'`). Assim, usar *raw strings* em expressões regulares evita problemas e torna o código mais legível e confiável.

[27]:
```
# Encotrar número 12 ou 123
re.findall(r'123?', '123 12 124 132')
```

[27]:
```
['123', '12', '12']
```

[28]:
```
# Listar caracteres diferentes de a, b e c que seguem o número 12
# Note que os parênteses neste exemplo servem para definir qual dígito será␣
 ↪listado
re.findall(r'12([^a-c])', '12a 12c 12d 125 13d')
```

[28]:
```
['d', '5']
```

[29]:
```
# Encontra números no texto
re.findall(r'[0-9]+', 'Em 2022 exitiam na Chesf em torno de 3200 funcionários.')
```

[29]:
```
['2022', '3200']
```

[30]:
```
# Encontrar e-mails em um texto
re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
           'Esta semana meu e-mail mudou de funcionario@empresa.gov.br para␣
 ↪funcionario@cempresa.com.br.')
```

[30]:
```
['funcionario@empresa.gov.br', 'funcionario@cempresa.com.br']
```

[31]:
```
# Encontrar mais 3 ou mais dígitos consecutivos que seja formado por 0, 1, 2, 3,␣
 ↪4 ou 5
re.search(r'[0-5]{3,}', 'adft12354879').group()
```

[31]:
```
'12354'
```

[32]:
```
# Encontrar mais 3 ou mais dígitos consecutivos que seja formados por números
re.search(r'\d{3,}', 'adft12354879').group()
```

[32]:
```
'12354879'
```

[33]:
```
# Encontrar mais 3 ou mais dígitos consecutivos que não seja formados por números
re.search(r'\D{3,}', 'adft12354879').group()
```

[33]:
```
'adft'
```

[34]:
```
# Separar texto usando caracteres entre a e f
re.split(r'[a-f]', '0a3b9z5f99p')
```

[34]:
```
['0', '3', '9z5', '99p']
```

[35]:
```
# Separar texto usando string formado de caracteres entre a e f
re.split(r'[a-f]+', '0ad3kbd9t')
```

[35]:
```
['0', '3k', '9t']
```

[36]:
```
re.sub(r':|;|,', '.', '0:3;9,10')
```

[36]:
```
'0.3.9.10'
```

[37]:
```
re.sub(r'\.+', '.', '0.....3...9..10..............5')
```

[37]:
```
'0.3.9.10.5'
```

Pode-se tratar textos lidos de um arquivo em padrão texto ou mesmo de uma lista de um arquivo MS Excel com muitas células e criação de uma nova coluna listando os textos tratados.

3.2.3 Listas

A forma mais comum de tratar listas no Python é utilizando a classe `list`. Muitos pacotes usam como base o tipo de lista `ndarray` do pacote NumPy em computação científica. Os objeto `ndarray` e `list` tem comportamentos e utilizações bem distintas. Nesta seção veremos funcionalidades básicas da classe `list` do Python e abordaremos listas do tipo `ndarray` quando falarmos do NumPy .

Métodos de uma Listas

append	index	sort
clear	insert	
copy	pop	
count	remove	
extend	reverse	

A forma básica de utilizar uma lista em Python é usar chaves para delimitar o início e fim da lista tendo seus elementos separados por vírgula. A lista pode contar elementos de diversos tipos, inclusive outras listas.

[38]:
```
l1 = [1, [2, 'three'], 4.5]  # Lista com número inteiro, float e outra lista
l2 = range(2, 20, 2)  # Lista de números inteiros no intervalo [2,20) com␣
 ↪intervalo de 2 elementos
print(l1, '\n')
print(l2, '\n')
print(list(l2), '\n')
```

```
[1, [2, 'three'], 4.5]

range(2, 20, 2)

[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

[39]:
```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
```

[40]:
```
lista.append('c')  # Adicionar "c" na lista
lista
```

[40]: ['a', 'b', 'c', 'd', 'e', 'f', 'c']

[41]:
```
lista.count('c')  # Contar "c" na lista
```

[41]: 2

[42]:
```
lista.extend([3,'Python',3.14])  # Expandir lista com outra lista
lista
```

[42]: ['a', 'b', 'c', 'd', 'e', 'f', 'c', 3, 'Python', 3.14]

[43]:
```
print(lista)
lista.pop(-1)  # Retirar item de lista por posição (último elemento)
lista.pop(2)  # Retirar item de lista por posição (terceito elemento)
lista.remove('a')
lista.remove(3)
print(lista)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'c', 3, 'Python', 3.14]
['b', 'd', 'e', 'f', 'c', 'Python']
```

[44]:
```
a = [3.14, 5.1, 1.73, 9, 4, 2.7182]
a.sort()
a
```

[44]: [1.73, 2.7182, 3.14, 4, 5.1, 9]

A biblioteca padrão do Python conta com o módulo `random` com algumas funções básicas de geração de números randômicos (ou pseudo randômicos como os mais rigorosos gostam de definir), bem como escolha aleatória em uma lista de números, mas existe a biblioteca `random` do NumPy com muito mais recursos disponíveis.

Funções disponíveis na biblioteca `random` padrão do Python.

betavariate	getstate	sample
binomialvariate	lognormvariate	seed
choice	normalvariate	setstate
choices	paretovariate	shuffle
expovariate	randbytes	triangular
gammavariate	randint	uniform
gauss	random	vonmisesvariate
getrandbits	randrange	weibullvariate

[45]:
```
import random

random.random()  # Número float randômico
```

[45]:
```
0.139485974955325
```

[46]:
```
b =['abacate', 'banana', 'côco', 'damasco', 'embaúba', 'figo']

random.shuffle(b) # Misturas a lista b
print(b)

b.sort()  # Ordenar a lista "b"
print(b)

print(random.choice(b))  # Escolhe um dos elementos da lista de forma aleatória

print(random.sample(b, 3))  # Escolhe uma amostra de 3 elementos
```

```
['côco', 'figo', 'abacate', 'embaúba', 'damasco', 'banana']
['abacate', 'banana', 'côco', 'damasco', 'embaúba', 'figo']
banana
['figo', 'damasco', 'embaúba']
```

[47]:
```
sorted(b, key=len)  # Ordenar a lista "b" com base no tamanho da palavra
```

[47]:
```
['côco', 'figo', 'banana', 'abacate', 'damasco', 'embaúba']
```

[48]:
```
[1,2,3] + [4,5,6]  # Juntar listas
```

[48]:
```
[1, 2, 3, 4, 5, 6]
```

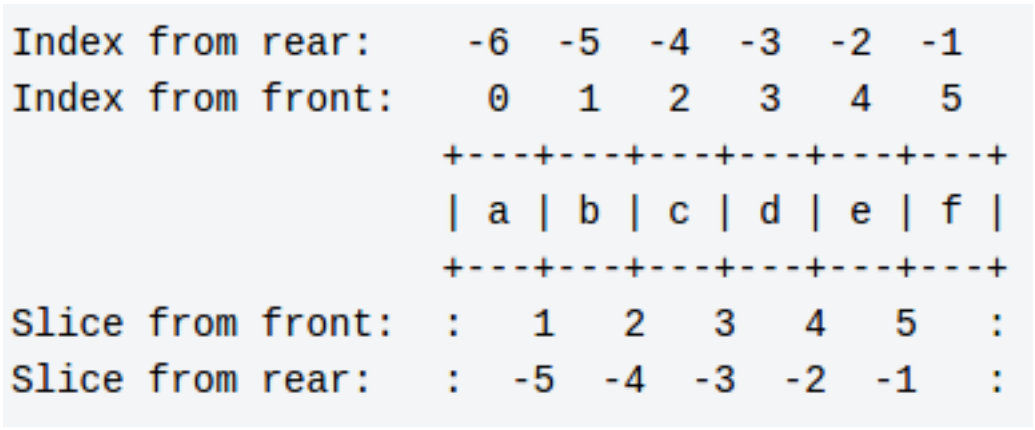
[49]:
```
3 * [1,2,3]  # Repetir listas
```

[49]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

A seleção de elementos de uma lista é uma funcionalidade importante e deve-se entender bem como funciona. O *slice* de uma lista tem a seguinte estrutura:

`Lista[de: até: passo]`

Na figura abaixo é detalhado como se pode selecionar elementos de uma lista. A contagem dos elementos inicia do zero e pode-se selecionar um elemento ou um intervalo de elementos. Para a seleção de intervalo ficar mais natural, pense que o intervalo é definido pelo índice entre os elementos e não o elemento em si, como sugere a figura.



Pode-se definir o índice do elemento da esquerda para direita utilizando números inteiros positivos ou da direita para esquerda utilizando números inteiros negativos.

[50]:
```
lista = ['a', 'b', 'c', 'd', 'e', 'f']
```

[51]:
```
lista[3]  # Quarto elemento
```

[51]: 'd'

[52]:
```
lista[-2]  # Penúltimo elemento
```

[52]: 'e'

[53]:
```
# Da posição entre b e c até a posição entre penúltimo e último elemento
lista[2:-1]
```

[53]: ['c', 'd', 'e']

[54]:
```
# Da posição entre a e b, até a posição entre e e f, de dois em dois elementos
lista[1:-1:2]
```

[54]: ['b', 'd']

Não colocar o número de índice da posição "de" do *slice* é equivalente a colocar zero. Da mesma forma não colocar o número de índice da posição "até" é equivalente a colocar a última posição da lista.

[55]:
```
print(lista[0:3])
print(lista[:3])  # Igual ao anterior
```

```
['a', 'b', 'c']
['a', 'b', 'c']
```

[56]:
```
print(lista[3:6])
print(lista[3:])  # Igual ao anterior
```

```
['d', 'e', 'f']
['d', 'e', 'f']
```

Uma forma fácil de inverter uma lista é pedir a lista do início ao final com passo -1.

[57]:
```
lista[::-1]   # Inverte lista, passo = -1
```

[57]:
```
['f', 'e', 'd', 'c', 'b', 'a']
```

No Python se poder fazer muita manipulação de lista em apenas uma linha, utilizando métodos sobre outros métodos ou, no caso deste exemplo, faze uma segunda seleção em cima de uma seleção já realizada. Cuidado para não cair na armadilha de ao enxugar o script não comprometer o entendimento, a facilidade de entender o código.

[58]:
```
lista[::-1][:-3]  # Da lista invertida, pegar até o antepenúltimo elemento
```

[58]:
```
['f', 'e', 'd']
```

Além da funcionalidade de seleção, a técnica de *slice* pode servir para alterar a lista.

[59]:
```
# Redefinindo calor de elementos
lista[0] = 'Primeiro'
lista[2:4] = ['Bola', 'Casa']
lista
```

[59]:
```
['Primeiro', 'b', 'Bola', 'Casa', 'e', 'f']
```

Temos agora um exemplo de modificação de um elemento de uma lista dentro de outra lista. Em `ldel[1][2]` definimos que deve ser selecionado o segundo elemento da primeira lista (índice 1) e deste elemento selecionado deve-se selecionar o terceiro elemento (índice 2).

[60]:
```
ldel = [['_', '_', '_', '_'],
              ['_', '_', '_'],
                   ['_', '_']]
ldel[1][2] = 'X'
ldel
```

[60]:
```
[['_', '_', '_', '_'], ['_', '_', 'X'], ['_', '_']]
```

A técnica de *slice* também funciona da mesma forma para variáveis do tipo `string` e `tuple`.

[61]:
```
s = '!!!   Python é de "torar"! Fácil, versátil e 100% grátis.   !!!'
print(s[6:12])
print(s[::-1])
```

```
Python
!!!   .sitárg %001 e litásrev ,licáF !"rarot" ed é nohtyP   !!!
```

[62]:
```
# Salvar os elementos da lista em variáveis independentes
a, b, c = ['Python', 'C++', 'javascript']
```

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

```
Python
C++
javascript
```

Uma funcionalidade muito útil nas listas é chamada **desempacotamento de variáveis** que é utilizada através do símbolo `*`. Abaixo temos um exemplo de junção de valores em uma variável. No exemplo usamos de forma separada as variáveis `a` e `b` e colocamos os demais valores na variável `c`.

[63]:
```
a, b, *c = range(5)  # * => as demais em "c"
print(a)
print(b)
print(c)
```

```
0
1
[2, 3, 4]
```

No próximo exemplo pegamos o primeiro valor e associamos a variável `a`, o último valor associado a variável `c` e os demais valores a variável `b`.

[64]:
```
a, *b, c = range(5)  # * => as demais em "b"
print(a)
print(b)
print(c)
```

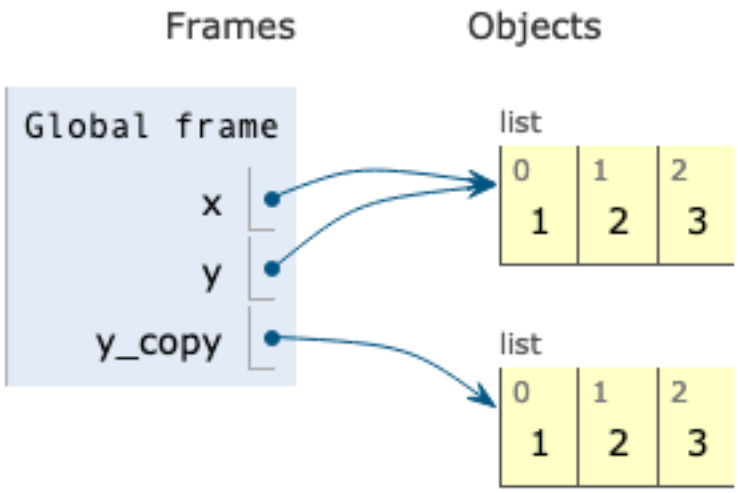
```
0
[1, 2, 3]
4
```

Podemos utilizas o `*` para desempacotar uma lista a ser inserida em uma função. No exemplo abaixo o comando `print(b)` tem como saída a lista `b`, mas de colocamos um asterisco antes da variável a saída é equivalente a `print(b[0], b[1], b[2])`.

[65]:
```
print(b)
print(*b)
print(b[0], b[1], b[2])
```

```
[1, 2, 3]
1 2 3
1 2 3
```

As variáveis devem ser pensadas como etiquetas de valores e não como caixas. As variáveis são referências que apontam para endereços de memória e não para um valor em si. Se o valor gravado em um endereço de memória é alterado, a variável tem seu valor alterado. O comportamento que vamos exemplificar ocorre com todas as variáveis mutáveis (listas, set e dicionários)



[66]:
```
x = [1, 2, 3]
print(f'x = {x}', '\n')

y = x
y_cópia = x.copy()

print(f'Identidade de x       = {id(x)}')
print(f'Identidade de y       = {id(y)}')
print(f'Identidade de y_cópia = {id(y_cópia)}', '\n')


print('Mudando x[0]', '\n')

x[0] = 5

print(f'x = {x}')
print(f'y = {y}')
print(f'y_cópia = {y_cópia}')
```

```
x = [1, 2, 3]

Identidade de x       = 1927724828416
Identidade de y       = 1927724828416
Identidade de y_cópia = 1927724828800

Mudando x[0]

x = [5, 2, 3]
y = [5, 2, 3]
y_cópia = [1, 2, 3]
```

[67]:
```
x = [1, 2, 3]
y = [1, 2, 3]

print(f'x = {x}')
print(f'y = {y}', '\n')

print(f'Identidade de x = {id(x)}')
print(f'Identidade de y = {id(y)}', '\n')

print('Mudando x[0]', '\n')

x[0] = 5

print(f'x = {x}')
print(f'y = {y}')
```

```
x = [1, 2, 3]
y = [1, 2, 3]

Identidade de x = 1927725006144
Identidade de y = 1927724827200

Mudando x[0]

x = [5, 2, 3]
y = [1, 2, 3]
```

[68]:
```
a  = 1  # Variável imutável
b  = 1  # Variável imutável

# Os valores de a e b são iguais e estão no mesmo local na memória
print('a is b: ', a is b)
print('a == b: ', a == b)
print()

a1 = [1]  # Variável mutável
b1 = [1]  # Variável mutável

# Os valores de a1 e b1 são iguais, mas não estão no mesmo local na memória
print('a1 is b1: ', a1 is b1)
print('a1 == b1: ', a1 == b1)
```

```
a is b:  True
a == b:  True

a1 is b1:  False
a1 == b1:  True
```

3.2.4 Tupla

Tuplas são listas imutáveis.

[69]:
```
tupla = (1, 'spam', 4, 'U')
print(tupla)
(type(tupla))
```

```
(1, 'spam', 4, 'U')
```

[69]:
```
tuple
```

A tupla pode ser mais eficiente que a lista do ponto de vista de utilização de memória.

3.2.5 Dicionário

[70]:
```
d = {'Python': 4, 'C++':5, 'R':0}
d
```

[70]:
```
{'Python': 4, 'C++': 5, 'R': 0}
```

[71]:
```
print(d.keys())
print(d.values())
```

```
dict_keys(['Python', 'C++', 'R'])
dict_values([4, 5, 0])
```

[72]:
```
d['Python']
```

[72]:
```
4
```

[73]:
```
d['Python'] += 1  # d['Python'] = d['Python'] + 1
d
```

[73]:
```
{'Python': 5, 'C++': 5, 'R': 0}
```

[74]:
```
d['Julia'] = 'nova'
d
```

[74]:
```
{'Python': 5, 'C++': 5, 'R': 0, 'Julia': 'nova'}
```

Métodos de dicionário

clear	pop
copy	popitem
fromkeys	setdefault
get	update
items	values
keys	

[75]:
```
print(d.get('Python', 'Não achei')) # Se chave Python existir, senão 'Não achei'
print(d.get('Java', 'Não achei'))   # Se chave Java existir, senão 'Não achei'
```

```
5
Não achei
```

3.2.6 Conjunto (*set*)

Conjunto	Python
$A \backslash B$	`A - B`
$A \cup B$	`A | B`
$A \cap B$	`A & B`
$A \subset B$	`A < B`
$A \triangle B$	`A ^ B`
$e \in B$	`e in B`

Em notebooks do Colab ou Jupyter podem ser utilizados símbolos matemáticos utilizando o padrão do $\LaTeX$

[76]:
```
A = {'a', 'b', 'c', 'd', 1 , 2, 3, 4}
B = {'c', 'd', 'e', 'f', 3, 4, 5, 6}
C = {'c', 5}

print(type(A))

print(f'A não em B: {A-B}')
print(f'A união B: {A | B}')
print(f'A intersecção B: {A & B}')
print(f'C está contido em  A: {C < A}')
print(f'C está contido em  B: {C < B}')
print(f'ou em A ou em B: {A ^ B}')
print(f'3 pertence a B: {3 in B}')
```

```
<class 'set'>
A não em B: {1, 2, 'a', 'b'}
A união B: {'e', 1, 2, 3, 4, 'd', 'f', 5, 6, 'a', 'b', 'c'}
A intersecção B: {3, 4, 'd', 'c'}
C está contido em  A: False
C está contido em  B: True
ou em A ou em B: {'e', 1, 'f', 5, 6, 2, 'a', 'b'}
3 pertence a B: True
```

Métodos de `set`

add	intersection	remove
clear	intersection_update	symmetric_difference
copy	isdisjoint	symmetric_difference_update

difference	issubset	union
difference_update	issuperset	update
discard	pop	

[77]:
```
lista = [1,2,3,4,3,2,3,4,5,3,1,5,6,4,2,7,3,2]
D = set(lista)
D
```

[77]:
```
{1, 2, 3, 4, 5, 6, 7}
```

3.3 Condicional

A indentação é uma característica importante no Python, pois de acordo com a indentação se define o que está dentro da declaração condicional. Não se utiliza marcadores como `begin` e `end` ou outro delimitador como chaves.

A indentação também é utilizada em Loops e funções.

A estrutura condicional mais comumente utilizada no Python é `if`, seguindo a seguinte estrutura:

```
if <condição>:
    <expressões>
elif <condição>:
    <expressões>
.
.
.

else:
    <expressões>
```

[78]:
```
x = 3

if 1 <= x < 4: # x em [1,4)
    print('Dentro do intervalo')
elif x%2==0:
    print('Fora do intervalo, mas é par')
elif type(x)==int:
    print('Fora do intervalo, não é par, mas é inteiro')
else:
    print('Fora do intervalo e não é inteiro')
```

```
Dentro do intervalo
```

[79]:
```
x = 4

if 1 <= x < 4: # x em [1,4)
```

```
    print('Dentro do intervalo')
elif x%2==0:
    print('Fora do intervalo, mas é par')
elif type(x)==int:
    print('Fora do intervalo, não é par, mas é inteiro')
else:
    print('Fora do intervalo e não é inteiro')
```

```
Fora do intervalo, mas é par
```

[80]:
```
ls = []

if ls:
    print('Lista vazia')
else:
    ls.append(4)
    if ls:
        print(f'Agoooora: {ls}')
```

```
Agoooora: [4]
```

[81]:
```
a = 2
b = 4

if a%2==0 and b%2==0:
    print('Ambos pares')
```

```
Ambos pares
```

Para processa as condicionais, o Python considera lista, tupla e conjunto vazios como faltos, em como o número 0 e string sem caracter algum.

[82]:
```
print(f'[]: {bool([])!s:>10}')
print(f'(): {bool(())!s:>10}')
print(f'set(): {bool(set())!s:>7}')
print(f'0: {bool(0)!s:>11}')
print(f'"": {bool("")!s:>10}')
```

```
[]:       False
():       False
set():    False
0:        False
"":       False
```

A partir do Python 3.10 foi inserido mais uma função condicional, a função `match`, que segue a estrutura abaixo:

“‘python match : case <valor 1>: case <valor 2>:

.
.

```
.

case _ :  # Não é nenhum dos valores anteriores
    <expressões>
'''
```

[83]:
```
nome = 'Raniere'

match nome[-1]:  # última letra no nome
    case 'a':
        print('Nome feminino')
    case 'o':
        print('Nome masculino')
    case _:
        print('Não tenho certeza.')
```

```
Não tenho certeza.
```

Atribuição condicional de variável

[84]:
```
k = 1
x = 5 if k==1 else 4
print(x)

k = 0
x = 5 if k==1 else 4
print(x)
```

```
5
4
```

3.4 Laços de repetição (*Loops*)

Declarações	Utilização
`pass`	Reserva de espaço vazio
`break`	Saída de laço
`continue`	Continuar o laço

O `for` do Python funciona como o `foreach` existente em algumas linguagens de programação.

[85]:
```
for letra in ['a', 'b', 'c']:
    print(letra)
```

```
a
b
c
```

[86]:
```
for i in range(5):
    print(5*'-')
    print(f'Valor {i}')

print('Acabou o for')
```

```
-----
Valor 0
-----
Valor 1
-----
Valor 2
-----
Valor 3
-----
Valor 4
Acabou o for
```

Pode ser utilizado a palavra chave `else` na estrutura do `for`. As expressões definidas no `else` só são executadas se o comando `break` não foi executado, ou seja, que o `for` passou por todos os elementos da lista.

[87]:
```
letra_proibida = 'c'

for l in ['a', 'b', 'c', 'd', 'e']:
    if l != letra_proibida:
        print(f'{l}, não é o {letra_proibida}')
    else:
        print(f'Chegou o {letra_proibida}')
        break
else:  # só se não for dado o comando break
    print('Não precisei para')

print('E acabou-se')
```

```
a, não é o c
b, não é o c
Chegou o c
E acabou-se
```

[88]:
```
letra_proibida = 'f'

for l in ['a', 'b', 'c', 'd', 'e']:
    if l != letra_proibida:
        print(f'{l}, não é o {letra_proibida}')
    else:
        print(f'Chegou o {letra_proibida}')
        break
else:  # só se não for dado o comando break
```

```
    print('Não precisei para')

print('E acabou-se')
```

```
a, não é o f
b, não é o f
c, não é o f
d, não é o f
e, não é o f
Não precisei para
E acabou-se
```

[89]:
```
for n,c in zip([1,2,3], ['a', 'b','c']):  # Juntar duas listas no for
    print(f'Número {n} e letra {c}')
```

```
Número 1 e letra a
Número 2 e letra b
Número 3 e letra c
```

[90]:
```
for n,c in zip([1,2,3,4], ['a', 'b','c','d']):
    print(n*c)
```

```
a
bb
ccc
dddd
```

[91]:
```
# for dentro de for
for n in [1,2,3]:
    for c in ['a', 'b','c']:
        print(n*c)
```

```
a
b
c
aa
bb
cc
aaa
bbb
ccc
```

[92]:
```
for n,c in enumerate(['a', 'b','c', 'd', 'e']):  # Enumenração de elementos
    print(f'{n+1}) letra {c}')
```

```
1) letra a
2) letra b
3) letra c
4) letra d
5) letra e
```

Também existe no Python a estrutura de repetição `while`.

[93]:
```
k = 0
while k <= 5:  # Enquanto k menor ou igual a 5
    print(f'O número é {k}')
    k += 1
```

```
O número é 0
O número é 1
O número é 2
O número é 3
O número é 4
O número é 5
```

3.4.1 *List Comprehension* (Listcomps)

[94]:
```
# Lista com números de 0 a 9
lista = range(10)

# Para cada valor da lista, transformar em string e repetir 3 vezes
[str(i)*3 for i in lista]
```

[94]:
```
['000', '111', '222', '333', '444', '555', '666', '777', '888', '999']
```

[95]:
```
multi = []
for i in range(1,5):
    for j in range(11,15):
        multi.append(i*j)
multi
```

[95]:
```
[11, 12, 13, 14, 22, 24, 26, 28, 33, 36, 39, 42, 44, 48, 52, 56]
```

[96]:
```
# Equivalente a estrutura anterior
[i*j for i in range(1,5) for j in range(11,15)]
```

[96]:
```
[11, 12, 13, 14, 22, 24, 26, 28, 33, 36, 39, 42, 44, 48, 52, 56]
```

[97]:
```
par_pow = []
for i in lista:
    if i%2==0:
        par_pow.append(i**2)
par_pow
```

[97]:
```
[0, 4, 16, 36, 64]
```

[98]:
```
# Equivalente a estrutura anterior
[i**2 for i in lista if i%2==0]
```

[98]:
```
[0, 4, 16, 36, 64]
```

[99]:
```
# Produto Cartesiano de listas
[(i,j) for i in 'abcd'
       for j in range(4)]
```

[99]:
```
[('a', 0),
 ('a', 1),
 ('a', 2),
 ('a', 3),
 ('b', 0),
 ('b', 1),
 ('b', 2),
 ('b', 3),
 ('c', 0),
 ('c', 1),
 ('c', 2),
 ('c', 3),
 ('d', 0),
 ('d', 1),
 ('d', 2),
 ('d', 3)]
```

[100]:
```
# Dicionário
{k:v**2 for k,v in zip('abcdef', range(1,7))}
```

[100]:
```
{'a': 1, 'b': 4, 'c': 9, 'd': 16, 'e': 25, 'f': 36}
```

[101]:
```
# Tupla
tupla  = (abs(i) for i in range(-3,4))
print(type(tupla))
print(list(tupla))
```

```
<class 'generator'>
[3, 2, 1, 0, 1, 2, 3]
```

3.4.2 Estrutura de paradigma funcional com função map

[102]:
```
fx = lambda x: x**2 -3*x + 5

print(fx(10))

print(list(map(fx, [1,2,3,4,5])))
```

```
75
[3, 3, 5, 9, 15]
```

[103]:
```
def tratar(s):
    s = str(s).upper()
    s = s.replace('4', 'X')
```

```
    return 'EQ: ' + s

print(tratar('04de'))

entrada = ['05c1', '03T2', '04p2', '14d1']
print(list(map(tratar, entrada)))
```

```
EQ: 0XDE
['EQ: 05C1', 'EQ: 03T2', 'EQ: 0XP2', 'EQ: 1XD1']
```

3.5 Tratamento de erro (try/except/else)

[104]:
```
try:
    print('Vamos ver se Python é bom mesmo')
    print(2 * 'Python')
except:
    print('Aí tais querendo muito')
else:
    print('Deu certo')

print('Teste finalizado')
```

```
Vamos ver se Python é bom mesmo
PythonPython
Deu certo
Teste finalizado
```

```
>>> 2 + 'Python'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-e645bad84159> in <module>
----> 1 2 + 'Python'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

[105]:
```
try:
    print('Vamos ver se Python é bom mesmo')
    print(2 + 'Python')
except:
    print('Aí tais querendo muito')
else:
    print('Deu certo')

print('Teste finalizado')
```

```
Vamos ver se Python é bom mesmo
Aí tais querendo muito
Teste finalizado
```

3.6 Funções

[106]:
```
def func1(a=2, b=3):
    return a**b

print(func1(3,4))       # 3**4
print(func1())          # 2**3
print(func1(b=2))       # 2**2
print(func1(3))         # 3**3
print(func1(b=3,a=4))   # 4**3
```

```
81
8
4
27
64
```

[107]:
```
def func2(a, b, *c):
    print(a,b)
    print(c)
    return a**b + sum(c)

func2(2, 3, 4, 5, 6)  # 2**3 + (4+5+6)
```

```
2 3
(4, 5, 6)
```

[107]:
```
23
```

[108]:
```
def factorial(n):
    if n<2:
        return 1
    else:
        return n * factorial(n-1)

factorial(5)
```

[108]:
```
120
```

4 NumPy

O Python é uma linguagem de programação de aplicação geral, não é uma linguagem desenvolvida originalmente para utilização em computação científica. Se valendo da característica de relativa facilidade de criação de pacotes para o Python aplicando linguagens como C/C++ e Fortran, o NumPy foi desenvolvido com intuito de facilitar a computação científica no Python de forma performática. Desde a segunda metade da década de 2000 se tornou um pacote fundamental para computação científica em Python.

O NumPy é utilizado como base de outros importantes pacotes utilizados em computação científica

como Matplotlib, SciPy , pandas, TensorFlow, Scikit-Learn , Statsmodels, CVXPy, PyWavelets, entre outros. São disponibilizadas funções pré-compiladas em C, C++ e Fortran, muitas provenientes de pacotes matemáticos já consolidados como BLAS e LAPACK.

Não vamos abordar neste material o Scipy, mas este pacote é importante na área de computação científica com várias bibliotecas como por exemplo scipy.integrate (Integração e *Ordinary Differential Equations* - ODEs), scipy.interpolate (Interpolação), scipy.optimize (Otimização e zeros da função), scipy.signal (Processamento de Sinais) e scipy.stats (funções Estatísticas).

Os dois grandes diferenciais do NumPy são o objeto `ndarray` e as funções do tipo `ufunc`. O `ndarray` (*N-dimentional array*) é um objeto que representa um array multidimensional, com tipagem homogênea e com itens de tamanho fixo na memória. As `ufunc` (*Universal Function*) processam `ndarray`, evitado utilização de estrutura de laços, otimizando a execução do código.

Aos que conhecem Matlab e R, podem usar as referências NumPy for MATLAB users e NumPy for R (and S-Plus) users respectivamente.

Alguns conjuntos de funções disponíveis

Pacote	Descrição
numpy.polynomial	Polinômios
numpy.linalg	Álgebra Linear
numpy.random	Amostras Randômicas
numpy.fft	Transformada Discreta de Fourier

Executar no ipython/Jupyter/Colab `%pylab` ou `from pylab import *` em script Python é equivalente a importações de 27 módulos (entre eles `numpy` como `np`, `matplotlib.pyplot` como `plt`, `numpy.random` como `ramdom`, `numpy.fft` como `fft`, `numpy.linalg` como `linalg`, entre outros). Serial algo como:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import random
from numpy import linalg
.
.
.
```

Além disso, são importados quase 900 funções e constantes para serem utilizadas diretamente. São importadas constantes (`pi`, `e`, `Inf`, `NaN`), funções trigonométricas (`sin`, `sinh`, `arcsin`, `deg2rad`, etc), estatístivas e probabilidade (`mean`, `median`, `std`, `cov`, `rand`, `randn`, `choice`, `poisson`, etc), álgebra linear e manipulação de matrizes (`det`, `inv`, `solve`, `tensorinv`, etc), polinômios (`poly`, `root`, `polyfit`, etc). Seria algo equivalente a:

```
from matplotlib.pyplot import *
from numpy import *
from numpy.fft import *
from numpy.linalg import *
from numpy.polynomial import *
from numpy.random import *
```

.
.
.

O console do ipython pode ser executado como comando `ipython --pylab` e o console já abre como se o comando `from pylab import *` tivesse sido executado.

[109]:
```
from pylab import *
```

[110]:
```
# Versão do NumPy utilizada neste material
np.__version__
```

[110]:
```
'2.1.3'
```

[111]:
```
import sys
x1 = linspace(0, 2*pi, 128)  # Array de 0 a 2pi com 128 amostras
x2 = [float(i) for i in x1]  # Lista baseado em x1 com elementos float
print('x1:', type(x1), 'com elementos do tipo ', x1.dtype)
print('x2:', type(x2), 'com elementos do tipo ', type(x2[0]))
```

```
x1: <class 'numpy.ndarray'> com elementos do tipo  float64
x2: <class 'list'> com elementos do tipo  <class 'float'>
```

`ndarray` é mais eficiente utilizando operações de forma vetorial que `list` utilizando loops. No exemplo abaixo a operação de elevar 10.000 elementos de um `ndarray` é medido em µs (microsegundos) e a mesma operação utilizando `list` é medida em ms (milisegundos), lembrando que 1.000 µs é equivalente a 1 ms.

[112]:
```
x1 = linspace(0, 10*pi, 10000)  # Array de 0 a 10pi com 10.000 amostras
x2 = [float(i) for i in x1]     # Lista baseado em x1 com elementos float

print(f'Tipo de x1: {type(x1)}')
print(f'Tipo de x2: {type(x2)}')
```

```
Tipo de x1: <class 'numpy.ndarray'>
Tipo de x2: <class 'list'>
```

[113]:
```
%timeit x1**2
```

```
8.74 µs ± 1.18 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[114]:
```
def fsqrt(x):
    list_temp = []
    for i in x:
        list_temp.append(i**2)
    return list_temp

%timeit fsqrt(x2)
```

```
3.03 ms ± 662 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Existem várias formas de criar ndarray no NumPy.

[115]:
```
# Criando array
a = array([[10, 2, 1],
           [1, 5, 1],
           [2, 3, 10]])  # ndarray 3x3
b = arange(0, 20, 0.5).reshape(8, 5) # (8, -1) seria calculado as 5 colunas
c = linspace(0, 2*np.pi, 32)  # 32 números entre 0 e 2π
d = ones([3,3], dtype=complex)  # dtype poderia ser usado nas outras técnicas
```

[116]:
```
print(a)
print()
print(b)
print()
print(c)
print()
print(d)
```

```
[[10  2  1]
 [ 1  5  1]
 [ 2  3 10]]

[[ 0.   0.5  1.   1.5  2. ]
 [ 2.5  3.   3.5  4.   4.5]
 [ 5.   5.5  6.   6.5  7. ]
 [ 7.5  8.   8.5  9.   9.5]
 [10.  10.5 11.  11.5 12. ]
 [12.5 13.  13.5 14.  14.5]
 [15.  15.5 16.  16.5 17. ]
 [17.5 18.  18.5 19.  19.5]]

[0.         0.2026834  0.40536679 0.60805019 0.81073359 1.01341699
 1.21610038 1.41878378 1.62146718 1.82415057 2.02683397 2.22951737
 2.43220076 2.63488416 2.83756756 3.04025096 3.24293435 3.44561775
 3.64830115 3.85098454 4.05366794 4.25635134 4.45903473 4.66171813
 4.86440153 5.06708493 5.26976832 5.47245172 5.67513512 5.87781851
 6.08050191 6.28318531]

[[1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j]]
```

Além dos métodos mostrados para criar `ndarray` podemos carregar ou salvar dados em arquivos em formato específico do NumPy/Python (load/save) ou de arquivos texto (loadtxt/savetxt). Também é possível criar `dnarray` proveniente de arquivo no padrão do Matlab, mas utilizando função do pacote SciPy (scipy.io.loadmat).

Como exemplo vamos gravar o `ndarray` de `b` em um arquivo "matr.dat" e recarregar.

[117]:
```
savetxt('matr.dat', b)

b1 = loadtxt('matr.dat')

print(b1)  # Mostrar os valores do ndarray b1
print()
print(f'Tipo de dados em b1: {b1.dtype}')  # Tipo dos dados salvo no ndarray b1
```

```
[[ 0.   0.5  1.   1.5  2. ]
 [ 2.5  3.   3.5  4.   4.5]
 [ 5.   5.5  6.   6.5  7. ]
 [ 7.5  8.   8.5  9.   9.5]
 [10.  10.5 11.  11.5 12. ]
 [12.5 13.  13.5 14.  14.5]
 [15.  15.5 16.  16.5 17. ]
 [17.5 18.  18.5 19.  19.5]]

Tipo de dados em b1: float64
```

As listas do tipo **ndarray** têm elementos do mesmo tipo e este tipo é normalmente definido durante a criação da lista, mas podemos forçar um tipo para os elementos para, por exemplo, ajustarmos o uso de memória mais adequado em nosso script.

[118]:
```
# ndarray com inteiros de 8 bits (inteiros de -128 a 127)
a1 = array([[10, 2, 1, 5, 20],
            [1, 5, 1, 20, 18],
            [2, 3, 10, 8, 40],
            [7, 2, 50, 2, 50],
            [0, 8, 15, 9, 3]], dtype=int8)

a2 = array([[10, 2, 1, 5, 20],
            [1, 5, 1, 20, 18],
            [2, 3, 10, 8, 40],
            [7, 2, 50, 2, 50],
            [0, 8, 15, 9, 3]])

a1s = sys.getsizeof(a1)  # Memória utilizada em x1
a2s = sys.getsizeof(a2)  # Memória utilizada em x2
print(f'a1 com elementos {a1.dtype} está usando {a1s:_d} bytes')
print(f'a2 com elementos {a2.dtype} está usando {a2s:_d} bytes')
print(f'a2 está usando {a2s/a1s:.2f} mais memória que a1')
```

```
a1 com elementos int8 está usando 153 bytes
a2 com elementos int64 está usando 328 bytes
a2 está usando 2.14 mais memória que a1
```

Deve haver cautela, pois uma vez definido **ndarray** com inteiro de 8 bits não se consegue guardar inteiros maiores que a memória definida para a variável (neste caso números for do intervalo -128 e 127).

Para inteiros podem ser definidos `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` e `int64` e para números reais `float16`, `float32`, `float64` e `float128` (quando suportado). Os `dtypes` que iniciam com `u` são *unsigned*, ou seja, apenas números positivos. O `int8`aceita números entre -128 e 127, os `uint8`números entre 0 e 255, por exemplo.

[119]:
```
print(np.iinfo(uint8))  # Inteiro positivos de 8 bits
print(np.iinfo(int8))
print(np.iinfo(int16))
print(np.iinfo(int32))
print(np.iinfo(int64))

print('\n\n')

print(np.finfo(float16))
print(np.finfo(float32))
print(np.finfo(float64))
```

```
Machine parameters for uint8
---------------------------------------------------------------
min = 0
max = 255
---------------------------------------------------------------

Machine parameters for int8
---------------------------------------------------------------
min = -128
max = 127
---------------------------------------------------------------

Machine parameters for int16
---------------------------------------------------------------
min = -32768
max = 32767
---------------------------------------------------------------

Machine parameters for int32
---------------------------------------------------------------
min = -2147483648
max = 2147483647
---------------------------------------------------------------

Machine parameters for int64
---------------------------------------------------------------
min = -9223372036854775808
max = 9223372036854775807
---------------------------------------------------------------
```

```
Machine parameters for float16
---------------------------------------------------------------
precision =   3   resolution = 1.00040e-03
machep =    -10   eps =        9.76562e-04
negep =     -11   epsneg =     4.88281e-04
minexp =    -14   tiny =       6.10352e-05
maxexp =     16   max =        6.55040e+04
nexp =        5   min =        -max
smallest_normal = 6.10352e-05   smallest_subnormal = 5.96046e-08
---------------------------------------------------------------

Machine parameters for float32
---------------------------------------------------------------
precision =   6   resolution = 1.0000000e-06
machep =    -23   eps =        1.1920929e-07
negep =     -24   epsneg =     5.9604645e-08
minexp =   -126   tiny =       1.1754944e-38
maxexp =    128   max =        3.4028235e+38
nexp =        8   min =        -max
smallest_normal = 1.1754944e-38   smallest_subnormal = 1.4012985e-45
---------------------------------------------------------------

Machine parameters for float64
---------------------------------------------------------------
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.2204460492503131e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
smallest_normal = 2.2250738585072014e-308   smallest_subnormal =
4.9406564584124654e-324
---------------------------------------------------------------
```

O `ndarray` é um objeto multidimensional que pode representar lista no espaço $\mathbb{R}^n$. Podemos representar vetores $A_{(i)}$ com uma dimensão com i elementos, matrizes $A_{(i,j)}$ com $i \times j$ elementos e tensores com n dimensões.

Vetor (espaço vetorial $\mathbb{R}$)

$A_{(3)} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

Matriz (espaço vetorial $\mathbb{R}^2$)

$A_{(2,2)} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Tensor (espaço vetorial $\mathbb{R}^n$)

$$A_{(2,2,2)} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

4.1 Métodos de `ndarray`

Abaixo listamos os métodos de variáveis do tipo `ndarray` salientando em negrito alguns métodos muito utilizados.

T	clip	dot	itemset	prod	setflags	tobytes
all	compress	dtype	itemsize	ptp	**shape**	tofile
any	conj	dump	mT	put	size	tolist
argmax	conjugate	dumps	max	ravel	sort	tostring
argmin	copy	fill	**mean**	real	squeeze	trace
argpartition	ctypes	flags	min	repeat	**std**	transpose
argsort	**cumprod**	flat	nbytes	reshape	strides	var
astype	**cumsum**	flatten	ndim	resize	**sum**	view
base	data	getfield	newbyteorder	round	swapaxes	
byteswap	device	imag	nonzero	searchsorted	take	
choose	diagonal	item	partition	setfield	to_device	

[120]:
```
b
```

[120]:
```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5],
       [10. , 10.5, 11. , 11.5, 12. ],
       [12.5, 13. , 13.5, 14. , 14.5],
       [15. , 15.5, 16. , 16.5, 17. ],
       [17.5, 18. , 18.5, 19. , 19.5]])
```

Podemos aplicar os métodos em todos os elementos ou em um eixo específico.

[121]:
```
b.mean()  # Média de b
```

[121]:
```
np.float64(9.75)
```

No NumPy 2, houve mudanças na forma como os tipos numéricos são exibidos, e agora os valores aparecem no formato explícito da classe, como `np.float64(9.75)`, em vez de simplesmente `9.75`.O principal objetivo dessa alteração foi melhorar a clareza e a previsibilidade do tipo de dado, especialmente ao lidar com diferentes tipos numéricos do NumPy (`np.int32`, `np.float64`, etc.). Antes, ao imprimir um número do NumPy, ele podia ser exibido sem indicar seu tipo exato, o que levava a ambiguidades. Agora, o formato deixa claro que o valor pertence a um tipo específico do NumPy. A função `print` pode ser usada para termos o mesmo comportamento das versão anteriores do NumPy.

[122]:
```
print(b.mean())
```

```
9.75
```

[123]:
```
b.mean(axis=0) # Média das colunas
```

[123]:
```
array([ 8.75,  9.25,  9.75, 10.25, 10.75])
```

[124]:
```
b.mean(axis=1) # Média das linhas
```

[124]:
```
array([ 1. ,  3.5,  6. ,  8.5, 11. , 13.5, 16. , 18.5])
```

Por padrão a função numpy.std calcula desvio padrão populacional conforme fórmula abaixo.

$$\sigma = \sqrt{\frac{1}{N}\sum (x_i - \mu)^2}$$

[125]:
```
b.std(axis=1)  # Desvio padrão populacional de cada linha
```

[125]:
```
array([0.70710678, 0.70710678, 0.70710678, 0.70710678, 0.70710678,
       0.70710678, 0.70710678, 0.70710678])
```

Para calcular desvio padrão amostral (padrão do MS Excel, R, Julia e `python.statistics`) utilize atributo `ddof=1` :

$$s_x = \sqrt{\frac{1}{n-1}\sum (x_i - \bar{x})^2}$$

[126]:
```
b.std(1, ddof=1)  # Desvio padrão amostral de cada linha
```

[126]:
```
array([0.79056942, 0.79056942, 0.79056942, 0.79056942, 0.79056942,
       0.79056942, 0.79056942, 0.79056942])
```

[127]:
```
print(b, '\n')
print(b.cumsum(axis=0))  # Soma acumulada das colunas
```

```
[[ 0.   0.5  1.   1.5  2. ]
 [ 2.5  3.   3.5  4.   4.5]
 [ 5.   5.5  6.   6.5  7. ]
 [ 7.5  8.   8.5  9.   9.5]
 [10.  10.5 11.  11.5 12. ]
 [12.5 13.  13.5 14.  14.5]
 [15.  15.5 16.  16.5 17. ]
 [17.5 18.  18.5 19.  19.5]]

[[ 0.   0.5  1.   1.5  2. ]
 [ 2.5  3.5  4.5  5.5  6.5]
 [ 7.5  9.  10.5 12.  13.5]
 [15.  17.  19.  21.  23. ]
 [25.  27.5 30.  32.5 35. ]
 [37.5 40.5 43.5 46.5 49.5]]
```

```
 [52.5 56.  59.5 63.  66.5]
 [70.  74.  78.  82.  86. ]]
```

Também existem funcões importantes no próprio pacote `NumPy`, como a função `numpy.where` que altera o `ndarray` de forma condicional, seguindo a estrutura:

```
np.where(<condição>, <aplicar de condição verdadeira>, <aplicar se condição falsa>)
```

[128]:
```
a = np.arange(10)
a
```

[128]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[129]:
```
np.where(a < 5, a, 10*a)
```

[129]:
```
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

4.2 Indexing / Slicing

A seleção de elementos do `ndarray` funciona diferente do que vimos para o objeto `list` natural do Python.

[130]:
```
b
```

[130]:
```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [ 2.5,  3. ,  3.5,  4. ,  4.5],
       [ 5. ,  5.5,  6. ,  6.5,  7. ],
       [ 7.5,  8. ,  8.5,  9. ,  9.5],
       [10. , 10.5, 11. , 11.5, 12. ],
       [12.5, 13. , 13.5, 14. , 14.5],
       [15. , 15.5, 16. , 16.5, 17. ],
       [17.5, 18. , 18.5, 19. , 19.5]])
```

[131]:
```
b[0,:]  # Primeira linha
```

[131]:
```
array([0. , 0.5, 1. , 1.5, 2. ])
```

[132]:
```
b[:,1]  # Segunda coluna
```

[132]:
```
array([ 0.5,  3. ,  5.5,  8. , 10.5, 13. , 15.5, 18. ])
```

[133]:
```
b[1:3,1:3]  # Elementos b22, b23, b32 e b33
```

[133]:
```
array([[3. , 3.5],
       [5.5, 6. ]])
```

[134]:
```
b[:,[[1,4]]]  # Segunda e Quinta colunas
```

[134]:
```
array([[[ 0.5,  2. ]],

       [[ 3. ,  4.5]],

       [[ 5.5,  7. ]],

       [[ 8. ,  9.5]],

       [[10.5, 12. ]],

       [[13. , 14.5]],

       [[15.5, 17. ]],

       [[18. , 19.5]]])
```

[135]:
```
b[[0,-1]]  # Primeira e última linha
```

[135]:
```
array([[ 0. ,  0.5,  1. ,  1.5,  2. ],
       [17.5, 18. , 18.5, 19. , 19.5]])
```

Além de realizar seleção por indexação, podemos usar regras lógicas para realizar seleção de elementos.

[136]:
```
b>15
```

[136]:
```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False],
       [False,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True]])
```

[137]:
```
b[b>15]  # Elementos de b tal que elemento maior que 15
```

[137]:
```
array([15.5, 16. , 16.5, 17. , 17.5, 18. , 18.5, 19. , 19.5])
```

[138]:
```
print(b, '\n')
print(b[(b>1) & (b<10)], '\n')
print(b[(b>1) & (b<10)].sum())  # Soma dos números pertencentes a (1,10)
```

```
[[ 0.   0.5  1.   1.5  2. ]
 [ 2.5  3.   3.5  4.   4.5]
 [ 5.   5.5  6.   6.5  7. ]
 [ 7.5  8.   8.5  9.   9.5]
 [10.  10.5 11.  11.5 12. ]
 [12.5 13.  13.5 14.  14.5]]
```

```
 [15.  15.5 16.  16.5 17. ]
 [17.5 18.  18.5 19.  19.5]]

[1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5 8.  8.5 9.  9.5]

93.5
```

Assim como fizemos com `list`, podemos não apenas selecionar como modificar e filtrar os valores de elementos específicos do `ndarray`

[139]:
```
b[:,2] = b[:,2]**2  # Terceira coluna ao quadrado
b
```

[139]:
```
array([[  0.  ,   0.5 ,   1.  ,   1.5 ,   2.  ],
       [  2.5 ,   3.  ,  12.25,   4.  ,   4.5 ],
       [  5.  ,   5.5 ,  36.  ,   6.5 ,   7.  ],
       [  7.5 ,   8.  ,  72.25,   9.  ,   9.5 ],
       [ 10.  ,  10.5 , 121.  ,  11.5 ,  12.  ],
       [ 12.5 ,  13.  , 182.25,  14.  ,  14.5 ],
       [ 15.  ,  15.5 , 256.  ,  16.5 ,  17.  ],
       [ 17.5 ,  18.  , 342.25,  19.  ,  19.5 ]])
```

[140]:
```
b[b>50] = 0
b
```

[140]:
```
array([[ 0.  ,  0.5 ,  1.  ,  1.5 ,  2.  ],
       [ 2.5 ,  3.  , 12.25,  4.  ,  4.5 ],
       [ 5.  ,  5.5 , 36.  ,  6.5 ,  7.  ],
       [ 7.5 ,  8.  ,  0.  ,  9.  ,  9.5 ],
       [10.  , 10.5 ,  0.  , 11.5 , 12.  ],
       [12.5 , 13.  ,  0.  , 14.  , 14.5 ],
       [15.  , 15.5 ,  0.  , 16.5 , 17.  ],
       [17.5 , 18.  ,  0.  , 19.  , 19.5 ]])
```

4.3 Broadcasting

Broadcasting é o comportamento de trabalharmos com um tensor como se estivéssemos trabalhando com escalares, evitando termos que utilizar *loopings* para modificar cada elemento de um tensor.

[141]:
```
a
```

[141]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[142]:
```
5*a + a**2
```

[142]:
```
array([  0,   6,  14,  24,  36,  50,  66,  84, 104, 126])
```

[143]:
```
a + 5
```

[143]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Também é possível operação de um tensor no espaço $\mathbb{R}^n$ por outro tensor no espaço $\mathbb{R}^{n-i}$. No exemplo a seguir realizamos a multiplicação de uma matriz em $\mathbb{R}^2$ por um vetor em $\mathbb{R}$.

[144]:
```
a1 = array([[1, 2, 3],
            [4, 5, 6],
            [7, 8, 9]])

a2 = array([1.2, 2.3, 3.4])

a1 * a2 # Array (3,3) multiplicado por Array (1,3)
```

[144]:
```
array([[ 1.2,  4.6, 10.2],
       [ 4.8, 11.5, 20.4],
       [ 8.4, 18.4, 30.6]])
```

[145]:
```
print(a * a, '\n')  # Multiplicação elemento a elemento
print(a @ a, '\n')  # Multiplicação de matriz a por matriz a
print(a.dot(a))     # Equivalente ao código da linha anterior
```

```
[ 0  1  4  9 16 25 36 49 64 81]

285

285
```

[146]:
```
print(type(a))
print(type(np.sin))  # Universal function
print(np.sin(a),'\n')
print(np.rad2deg(np.sin(a)),'\n')
```

```
<class 'numpy.ndarray'>
<class 'numpy.ufunc'>
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
 -0.2794155   0.6569866   0.98935825  0.41211849]

[  0.          48.21273601  52.09890488   8.08558087 -43.36158891
 -54.94231381 -16.00932878  37.6425593   56.68605196  23.61264986]
```

O NumPy conta com muitas Funções Matemáticas como `ufunc`. Para mais informações verificar a documentação do NumPy disponível neste texto.

4.4 Mais Rotinas

[147]:
```
f = array([1,2,3,5,6,4,5,6,7,4,5,6,7,8,5,3,5,6,8,9,9,5,4,4])
```

[148]:
```
unique(f)
```

[148]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[149]:
```
histogram(f, bins=6)
```

[149]:
```
(array([ 2,  2,  4, 10,  2,  4]),
 array([1.        , 2.33333333, 3.66666667, 5.        , 6.33333333,
        7.66666667, 9.        ]))
```

[150]:
```
roots([  1,  -9,  26, -24])  # Raizes de x**3 - 9*x*2 + 26*x - 24
```

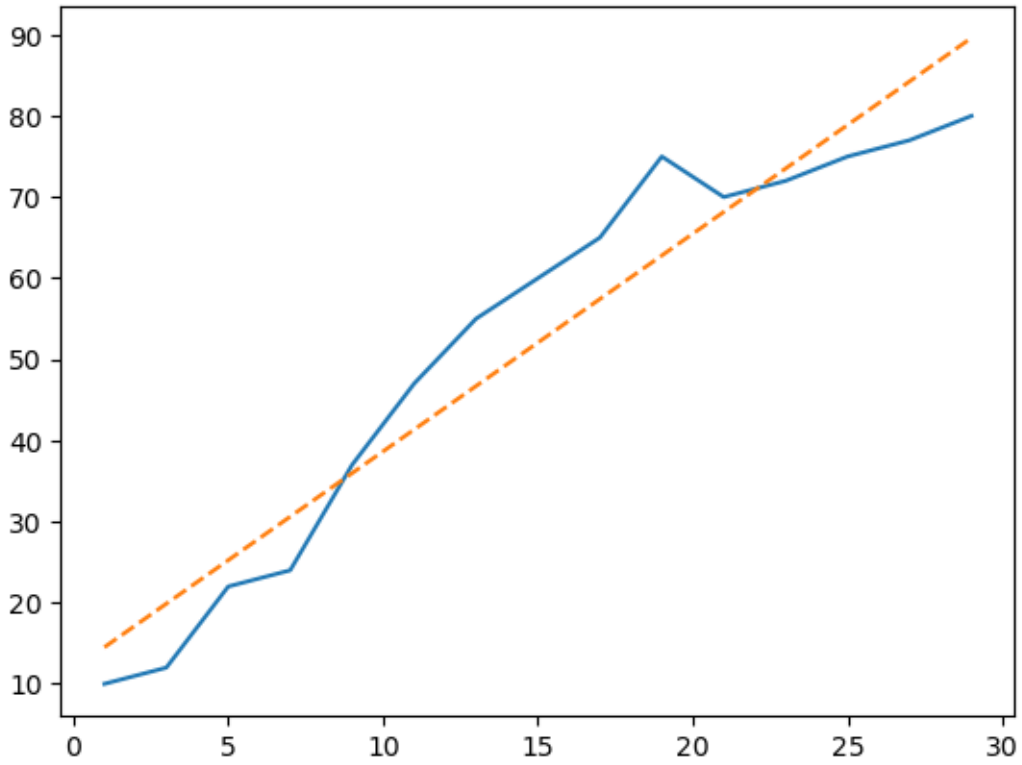
[150]:
```
array([4., 3., 2.])
```

[151]:
```
# Regressão Linear

x = [  1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
y = [ 10, 12, 22, 24, 37, 47, 55, 60, 65, 75, 70, 72, 75, 77, 80]

polyfit(x,y, 1) # Regressão em polinômio de ordem 1 (regressão linear)
```

[151]:
```
array([ 2.68214286, 11.83452381])
```

[152]:
```
a, b = polyfit(x,y, 1) # Coeficientes da regressão linear

plot(x,y)
plot(x, a*array(x) + b, '--')  # Linha de regressão
```

[152]:
```
[<matplotlib.lines.Line2D at 0x1c0d7ca3020>]
```



Foi utilizado o pacote Matplotlib para plotar o gráfico. Ainda vamos abortadr esta biblioteca a seguir.

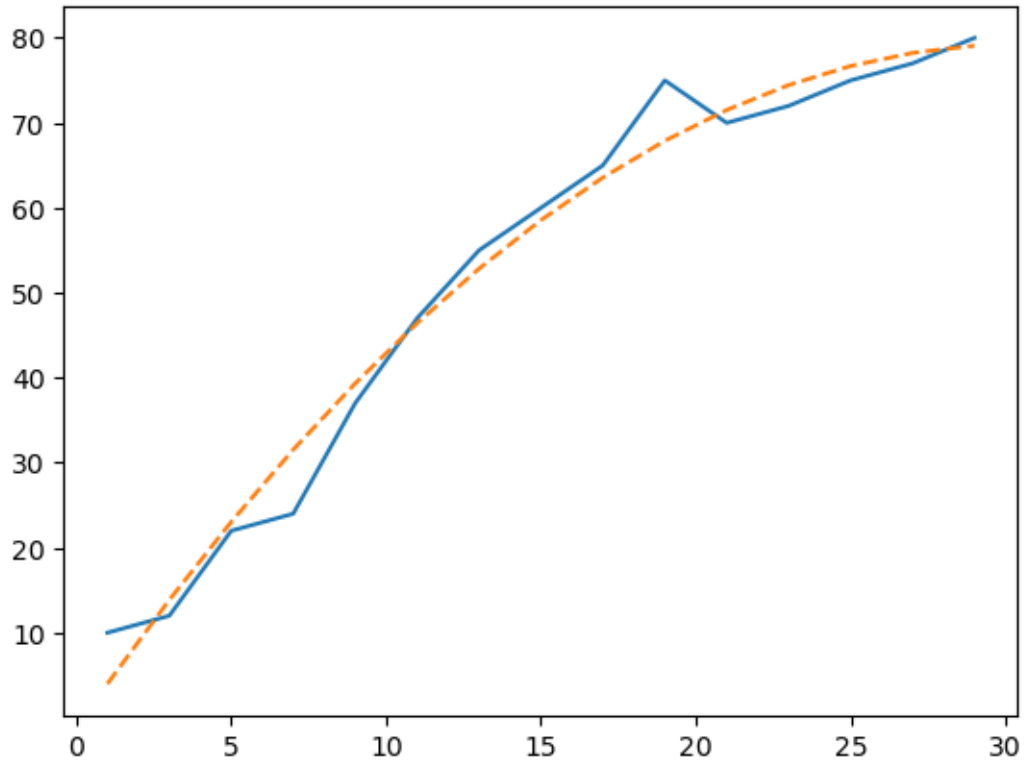
Abaixo temos o exemplo de uma regressão polinomial de ordem 2.

[153]:
```
polyfit(x,y, 2) # Regressão em polinômio de ordem 2
```

[153]:
```
array([-0.08682127,  5.28678087, -1.21760666])
```

[154]:
```
a, b, c = polyfit(x,y, 2) # Coeficientes da regressão linear

plot(x,y)
plot(x, a*array(x)**2 + b*array(x) + c, '--')  # Linha de regressão
```

[154]:
```
[<matplotlib.lines.Line2D at 0x1c0d8533770>]
```



4.5 Resolução de sistemas lineares com `numpy.linalg`

$$\begin{cases} 10x_1 + 0,5x_2 + 0,6x_3 + 3x_4 + 2x_5 + 3x_6 = 48,05 \\ 3x_1 + 1x_2 + 13x_3 + 5x_4 + 2x_5 + x_6 = 55 \\ x_1 + 10x_2 + 0,8x_3 + 2x_4 + 3x_5 + x_6 = 101 \\ 4x_1 + 2x_2 + x_3 + 15x_4 + 3x_5 + 4x_6 = 105 \\ x_1 + 0,5x_2 + 0,6x_3 + 0,3x_4 + 9x_5 + 5x_6 = 54,7 \\ 3x_1 + 2x_2 + 3x_3 + x_4 + 4x_5 + 15x_6 = 126 \end{cases}$$

$$A \times x = B$$

$$\begin{bmatrix} 10 & 0,5 & 0,6 & 3 & 2 & 3 \\ 1 & 10 & 0,8 & 2 & 3 & 1 \\ 3 & 1 & 13 & 5 & 2 & 1 \\ 4 & 2 & 1 & 15 & 3 & 4 \\ 1 & 0,5 & 0,6 & 0,3 & 9 & 5 \\ 3 & 2 & 3 & 1 & 4 & 15 \end{bmatrix} \times x = \begin{bmatrix} 48,5 \\ 55 \\ 101 \\ 105 \\ 54.7 \\ 126 \end{bmatrix}$$

[155]:
```
A = array([[10, 0.5, 0.6, 3, 2, 3],
           [1, 10, 0.8, 2, 3, 1],
           [3, 1, 13, 5, 2, 1],
           [4, 2, 1, 15, 3, 4],
```

```
            [1, 0.5, 0.6, 0.3, 9, 5],
            [3, 2, 3, 1, 4, 15]])

B = array([ 48.5,  55 , 101 , 105 ,  54.7, 126 ])

solve(A, B)
```

[155]:
```
array([1., 3., 5., 4., 2., 6.])
```

Salientamos que usamos diretamente da função `solve` por termos importado a função em `from pylab import *`. Se for utilizado a importação por `import numpy as np` o correto seria utilizada `np.linalg.solve(A, B)`.‘

$$x = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 4 \\ 2 \\ 6 \end{bmatrix}$$

[156]:
```
# Utilizando @ o Python entendo que é multiplicação de Matrizes com os ndarray
inv(A) @ B
```

[156]:
```
array([1., 3., 5., 4., 2., 6.])
```

`ndarray` não tem métodos de matrizes. Devem ser utilizadas funções do NumPy para realizar operações de matrizes com este tipo de array.

[157]:
```
det(A)  # Determinante da Matriz
```

[157]:
```
np.float64(1850722.1600000008)
```

[158]:
```
inv(A)  # Inversa da matriz
```

[158]:
```
array([[ 0.11370042,  0.0018882 , -0.00015139, -0.02183456, -0.01292944,
        -0.01272351],
       [-0.00331326,  0.10298361, -0.00565853, -0.011062  , -0.03212843,
         0.00783366],
       [-0.01525415, -0.00346535,  0.0783854 , -0.02291181, -0.00832108,
         0.00693967],
       [-0.02403062, -0.01087867, -0.00098783,  0.07417396, -0.01084161,
        -0.01056863],
       [-0.00097238,  0.00201067,  0.00396237, -0.00111338,  0.12925506,
        -0.04299184],
       [-0.01738614, -0.01322665, -0.01588311,  0.00577618, -0.02521135,
         0.07894801]])
```

[159]:
```
pinv(A)  # Moore-Penrose pseudo-inversa da matriz
```

[159]:
```
array([[ 0.11370042,  0.0018882 , -0.00015139, -0.02183456, -0.01292944,
        -0.01272351],
       [-0.00331326,  0.10298361, -0.00565853, -0.011062  , -0.03212843,
         0.00783366],
       [-0.01525415, -0.00346535,  0.0783854 , -0.02291181, -0.00832108,
         0.00693967],
       [-0.02403062, -0.01087867, -0.00098783,  0.07417396, -0.01084161,
        -0.01056863],
       [-0.00097238,  0.00201067,  0.00396237, -0.00111338,  0.12925506,
        -0.04299184],
       [-0.01738614, -0.01322665, -0.01588311,  0.00577618, -0.02521135,
         0.07894801]])
```

4.6 Matplotlib

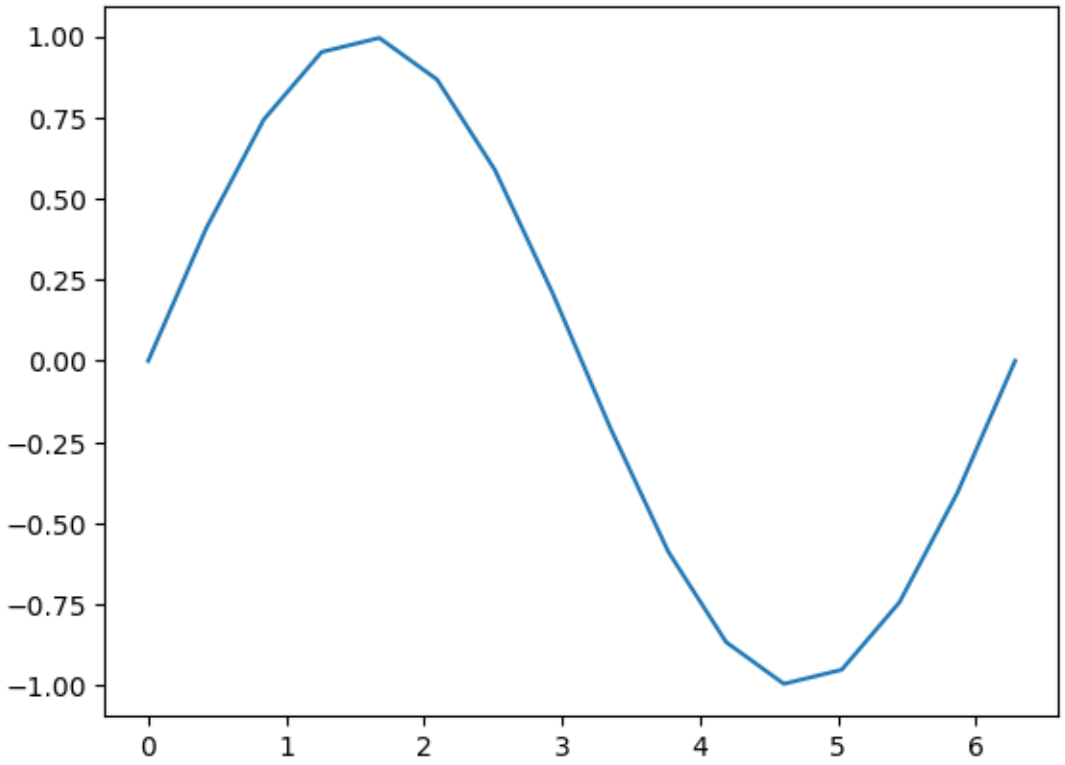
Uma das bibliotecas mais utilizadas no Python para criar visualizações de dados de forma estática, animada e interativa. Não temos intensão de esgotar todas as possibilidades do Matplotlib, mas passar pelo básico para plotar gráfico. Para aprofundamento a documentação oficial da ferramenta deve ser consultada.

[160]:
```
from pylab import *

matplotlib.__version__
```

[160]:
```
'3.10.0'
```

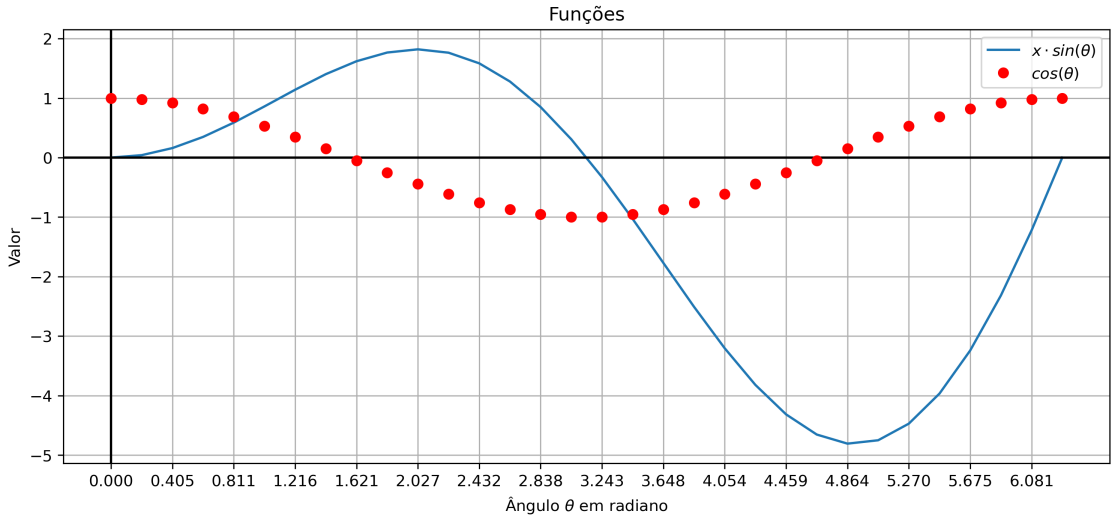
Abaixo realizamos a plotagem simples de um gráfico sendo x um `ndarray` com 16 pontos entre zero e 2π e y um `ndarray` com valores do seno de x.

[161]:
```
x = linspace(0, 2*pi, 16)  # Array no intervalo [0,2pi) dividido em 16 amostras
y = sin(x) # calcula sen de todo o dnarray x

plot(x,y)  # Plotar curva do sen
show()  # Mostrar gráfico
```



Vamos colocar mais algumas opções no gráfico.

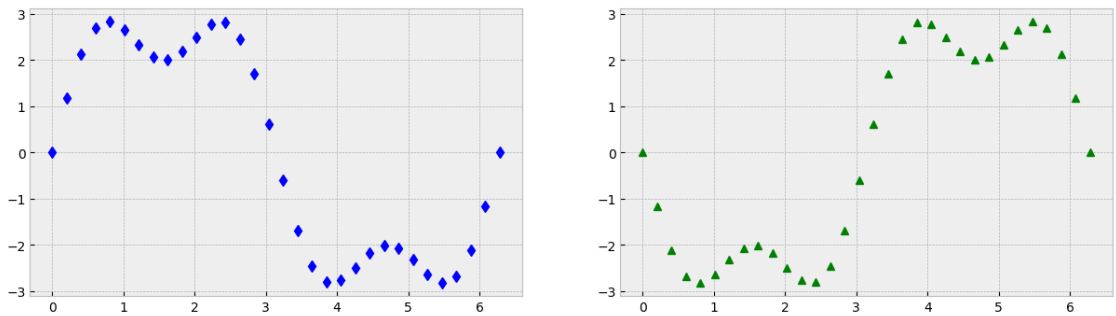
[162]:
```
θ = linspace(0, 2*pi, 32)  # Array no intervalo [0,2π) dividido em 32 amostras
ys = θ * sin(θ) # calcula seno de todo o dnarray, multiplicado pelo valor de θ
yc = cos(θ) # calcula cosseno de todo o dnarray θ

figure(figsize=(12,5), dpi=300)  # Redimensionar figura e aumentar resolução
plot(θ,ys, label='$x \\cdot sin(\\theta)$')   # Plotar curva do sen com legenda␣
 ↪em LaTeX
axhline(0, color='black') # Plotar eixo x na cor preta
axvline(0, color='black') # Plotar eixo y na cor preta
plot(θ,yc, 'ro', label='$cos(\\theta)$')  # Plotar curva do cos com legenda␣
 ↪"cos" utilizando marcador de ponto na cor vermelha
title('Funções')  # Definir título do gráfico
xticks(θ[::2])  # Colocar de 2 em 2 valores de θ no eixo x
xlabel('Ângulo $\\theta$ em radiano')  # Definir legenda do eixo x
ylabel('Valor')  # Definir legenda do eixo y
legend()  # Ativar legenda
grid()    # Ativar grid
show()    # Mostrar gráfico
```



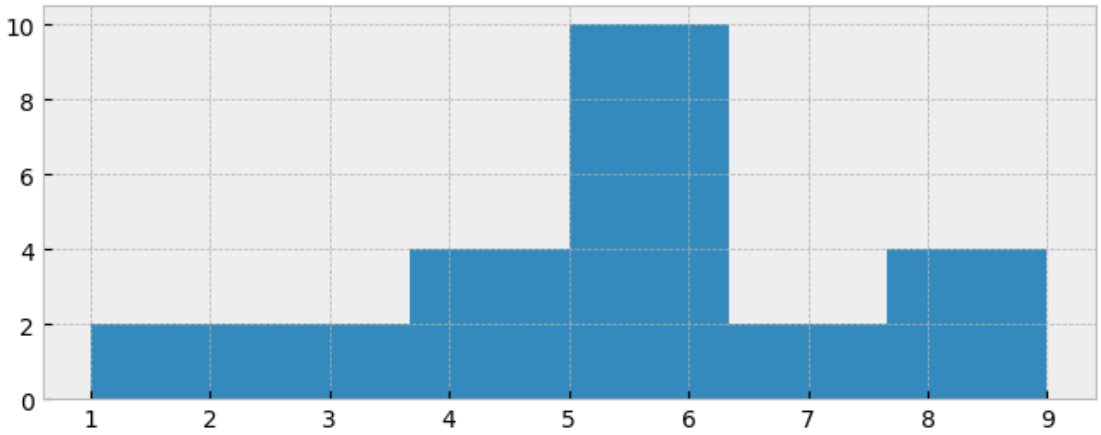
No exemplo acima, se quiser salvar a figura no lugar de mostrar gráfico em tela, deve-se substituir a linha `show()` por `savefig('Gráfico ds funções.png')`, que salvaria o gráfico em formato PNG com nome "Gráfico ds funções.png" no mesmo diretório onde está o script está rodando.

Podem ser definidos estilos distintos para mostrar o gráfico no matplotlib.

[163]:
```
plt.style.use('bmh')

x = linspace(0, 2*pi, 32)  # Array no intervalo [0,2pi) dividido em 32 amostras

def f(x): # definindo função a ser plotada
    return 3*np.sin(x) + np.sin(3*x)

# Criando dois gráficos em uma linha e duas colunas
fig, axs = subplots(1, 2, figsize=(15,4))
# Plotar função cor azul em formato de diamante
axs[0].plot(x,f(x), 'bd')
# Plotar função negativa cor azul em formato de triângulo
axs[1].plot(x,-f(x), 'g^')
show()
```



[164]:
```
f = array([1,2,3,5,6,4,5,6,7,4,5,6,7,8,5,3,5,6,8,9,9,5,4,4])
figure(figsize=(8,3))
hist(f, bins=6)
show()
```



4.7 Distribuições probabilísticas em `numpy.random`

[165]:
```
randint(100, size=(3, 5))  # Array com interos de 0 a 100 com dimensão 3x5b
```

[165]:
```
array([[78, 99, 14, 21, 24],
       [83, 68,  8, 10, 83],
       [61, 30, 55, 27, 48]], dtype=int32)
```

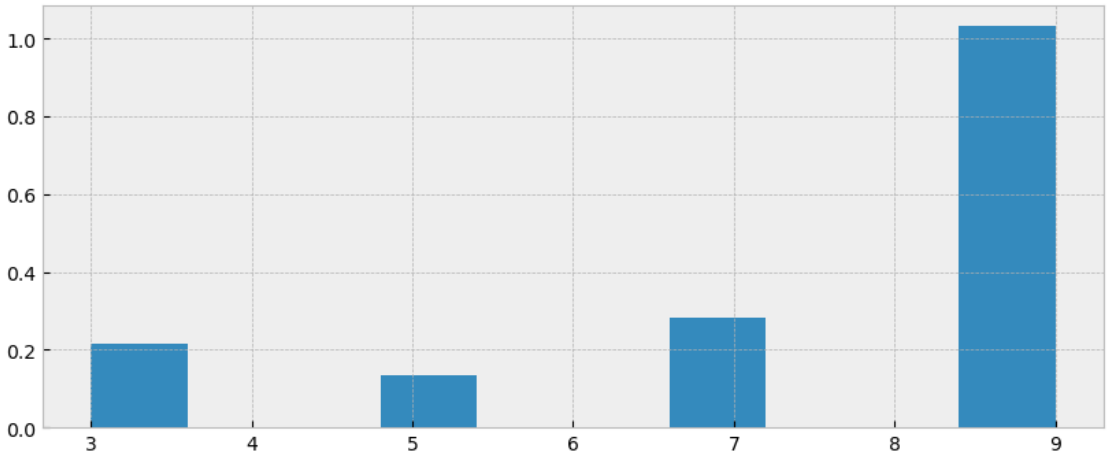
Lembrando que a utilização direta da função `randint` é possível por termos usado a importação por `from pylab import *`. Se for utilizado a importação por `import numpy as np` o correto seria utilizada `np.random.randint(100, size=(3, 5))`.

[166]:
```
rand(10)  # Array com 10 números aleatórios entre 0 e 1
```

[166]:
```
array([0.2355154 , 0.63026081, 0.80957689, 0.16145336, 0.33285827,
       0.08076332, 0.68131683, 0.60271951, 0.66322696, 0.21395939])
```

[167]:
```
# Escolha com probabilidade definida por elemento

# Dados de 100 números seguindo probailidade definida
# Números 3, 5 e 7 com probabilidade 0.15 e número 9 com prob. de 0.55
dados = np.random.choice([3, 5, 7, 9], p=(.15,.15,.15,.55), size=100)

plt.figure(figsize=(10,4))  # Definir tamanho do gráfico
```

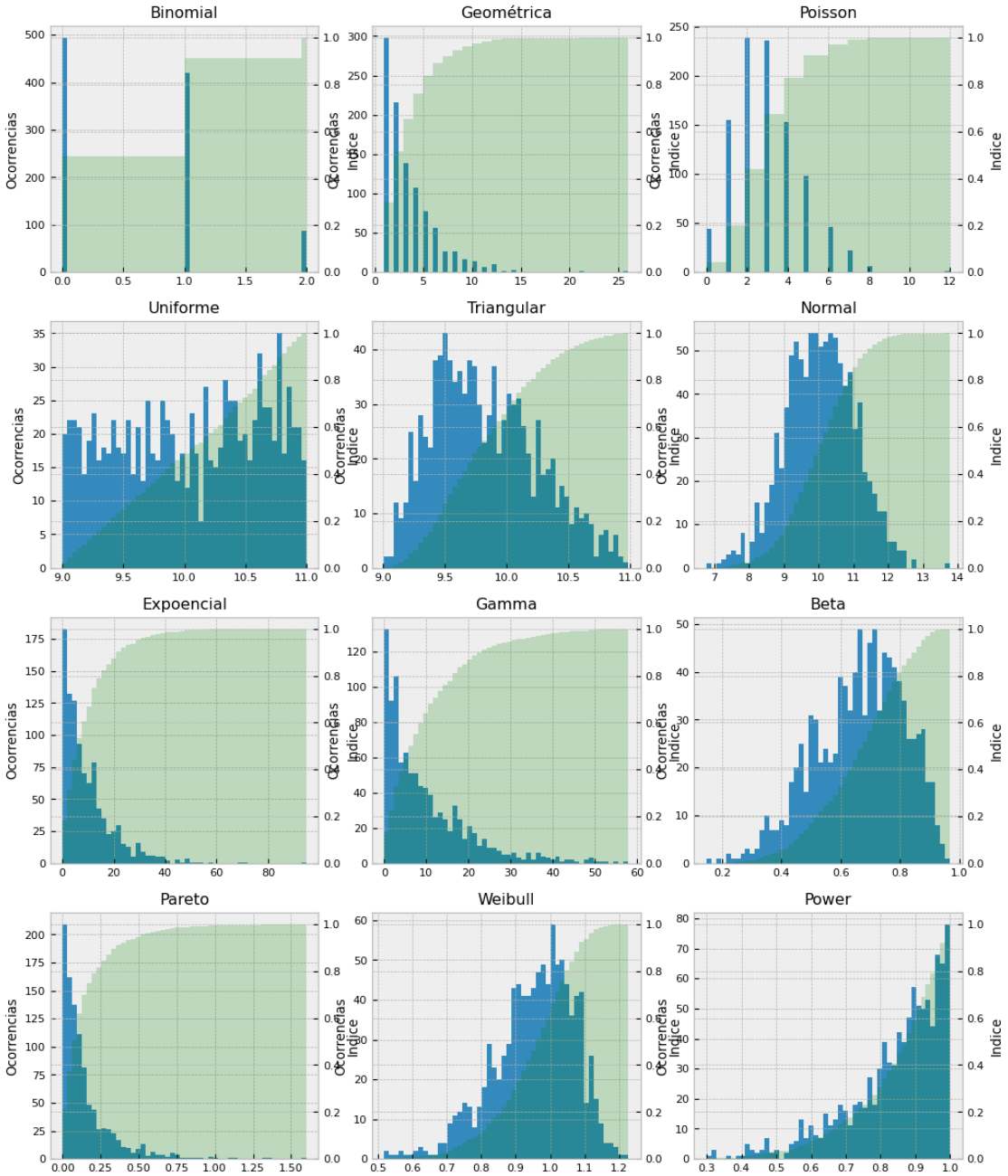
```
plt.hist(dados, density=True)  # Plotar histograma
plt.show()
```



O maior objetivo do exemplo abaixo é mostrar possibilidade de geração de distribuições probabilísticas, além de mais um exemplo de como podemos plotar gráficos usando `matplotlib`.

[168]:
```
bins = 50
media = 10
n = 1000 # Número de amostra

dc = dict()
dc['Binomial'] = binomial(2, 0.3, n)       # (tamanho, probabilidade, tamanho)
dc['Geométrica'] = geometric(0.3, n)       # (propabilidade, tamanho)
dc['Poisson'] = poisson(3, n)              # (lambda, tamanho)
dc['Uniforme'] = uniform(9,11,n)           # (mínimo, máximo, tamanho)
dc['Triangular'] = triangular(9,9.5,11,n) # (mínimo, moda, máximo, tamanho)
dc['Normal'] =  normal(media,1,n)          # (média, desvio padrão, tamanho)
dc['Expoencial'] =  exponential(media,n)   # (média, tamanho)
dc['Gamma'] =  gamma(1,media,n)            # (alpha, beta, tamanho)
dc['Beta'] = beta(6,3,n)                   # (alpha, beta, tamanho)
dc['Pareto'] = pareto(8,n)                 # (forma, tamanho)
dc['Weibull'] = weibull(10,n)              # (alpha, tamanho)
dc['Power'] = power(5,n)                   # (formato, tamanho)

fig = figure(figsize=(12,15))
rcParams['font.size'] = 8  # Definindo tamanho da fonte
for i, label in enumerate(dc.keys()):
    ax1 = fig.add_subplot(4, 3, i+1) # 4 linha, 3 colunas, sequencial
    ax1.hist(dc[label], bins, label=label)
    ax1.set_ylabel('Ocorrencias')
    ax2 = ax1.twinx()  # Criar segundo eixo de ordenada
```

```
    ax2.hist(dc[label], bins, density=True, cumulative=True,
             alpha=0.2, label='Acumulado', color='g')  # Histograma Cumulativo
    ax2.set_ylabel('Indice')
    title(label)

show()
```



5 Programação Orientada a Objeto

Não temos intenção de nos aprofundarmos em teoria de programação utilizando paradigma de Orientação a Objeto, mas sim explorar como podemos utilizamos esse paradigma em Python para escrever códigos. Podemos usar paradigma funcional, procedural ou orientação a objeto em Python, mas todas as variáveis criadas são sempre um objeto com instância de alguma classe pré-definida com seus atributos e métodos. Quando criamos uma variável com texto esta variável é uma instância da classe `str` com todos seus métodos e atributos.

[169]:
```
texto = 'Python'
print(type(texto))
```

```
<class 'str'>
```

A ferramenta utilizada em Python para implantar objetos é `class`. Abaixo vamos criar uma classe `Esfera` com atributos declarados `r` (raio) e `cor` (que terá a cor preta como pré-definida) e com métodos `area`(cálculo de área) e `volume` (cálculo de volume), sendo:

$$area = 4\pi r^2$$

$$volume = \frac{4}{3}\pi r^3$$

Para que os atributos calculados ou declarados estejam disponíveis para serem utilizados em qualquer parte da instância ou da classe o nome da variável sempre deve ser precedida de `self.` e em todos os métodos da classe o `self` sempre deve ser o primeiro atributo a ser declarado.

[170]:
```
from pylab import *

class Esfera:
    def __init__(self, r, cor='preta'):  # Método Dunder Construtor
        self.r = r  # Atributo declarado
        self.cor = cor  # Atributo declarado

    def area(self):  # Método
        return 4 * pi * self.r**2

    def volume(self):  # Método
        return 4/3 * pi * self.r**3
```

Sempre que uma impedância é criada o método `__init__` é executado. Este é o método em Python utilizado como **construtor** da classe.

[171]:
```
esfera1 = Esfera(4)

print(f'Área: {esfera1.area()}')
print(f'Volume: {esfera1.volume()}')
print(f'Cor: {esfera1.cor}')
```

```
Área: 201.06192982974676
Volume: 268.082573106329
Cor: preta
```

[172]:
```
esfera2 = Esfera(2, 'branca')

print(f'Área: {esfera2.area()}')
print(f'Volume: {esfera2.volume()}')
print(f'Cor: {esfera2.cor}')
```

```
Área: 50.26548245743669
Volume: 33.510321638291124
Cor: branca
```

Utilizando a função `dir` podemos listar todos os métodos e atributos de um objeto.

[173]:
```
dir(esfera2)
```

[173]:
```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'area',
 'cor',
 'r',
```

```
 'volume']
```

Note que para utilizar um método sempre se abre e fecha os parênteses após o nome do método, mesmo que não haja parâmetros a serem informador. No caso de utilização de um valor de atributo, como no caso de `esfera1.cor` os parênteses não são utilizados.

Em Python existem Métodos Especial que iniciam e terminam com "__", são os métodos Dunder, ou *Double Underscore Before and After*. Este tipo de método normalmente é utilizado com auxílio de operadores, como exemplificaremos mais à frente.

Temos mais um exemplo com classe `Impedancia` contendo atributos declarados R, L, C e f e com métodos `xc` (cálculo de impedância capacitiva), `xl` (impedância indutiva), `Z` (impedância) e `conteudo` (lista atributos e métodos, menos os especiais).

[174]:
```
class Impedancia:
    def __init__(self, R, L, C, f):  # Método Dunder Construtor
        self.R = R  # Atributo declarado
        self.__L = L  # Atributo declarado Encapsulado
        self.__C = C  # Atributo declarado Encapsulado
        self.f = f  # Atributo declarado
        self.ω = 2*pi*self.f  # Atributo calculado

    def xc(self):  # Método
        return (self.ω*self.__C)**-1

    def xl(self):  # Método
        return self.ω*self.__L

    def Z(self):  # Método
        X = (self.xl() - self.xc())
        return complex(self.R, X)

    def conteudo(self):
        return [i for i in dir(self) if not i.startswith('_')]
```

[175]:
```
Z = Impedancia(R=10, L=0.1, C=1e-3, f=60)  # Z: instância, Impedância: classe

print('Resistência: ',   Z.R)           # Atributo Resistência
print('Reatâmcia Capacitiva: ', Z.xc()) # Método cálculo da reatância capacitiva
print('Reatâmcia Indutiva: ', Z.xl())   # Método cálculo da reatância indutiva
print('Impedância: ', Z.Z())            # Método cálculo da impedância
```

```
Resistência:  10
Reatâmcia Capacitiva:  2.6525823848649224
Reatâmcia Indutiva:  37.69911184307752
Impedância:  (10+35.046529458212596j)
```

[176]:
```
Z1 = Impedancia(R=5, L=0.2, C=5e-3, f=50)
print('Resistência: ',   Z1.R)           # Atributo Resistência
```

```
print('Reatâmcia Capacitiva: ', Z1.xc()) # Método cálculo da reatância capacitiva
print('Reatâmcia Indutiva: ', Z1.xl())   # Método cálculo da reatância indutiva
print('Impedância: ', Z1.Z())
```

```
Resistência:  5
Reatâmcia Capacitiva:  0.6366197723675814
Reatâmcia Indutiva:  62.83185307179587
Impedância:  (5+62.19523329942829j)
```

[177]:
```
Z1.conteudo()
```

[177]:
```
['R', 'Z', 'conteudo', 'f', 'xc', 'xl', 'ω']
```

The Python Language Reference lista mais de 80 nomes de métodos especiais. Exemplo de alguns Dunders:

Método	Função
__init__	Executado no momento da criação a instância (método construtor)
__contains__	in
__eq__	==
__getitem__	minha_instância[x]
__call__	minha_instância(x)
__add__	+

[178]:
```
class Teste:
    def __init__(self, a):
        self.a = a

    def __eq__(self,b):
        print('ôxe, que pergunta da gota!!!')
        return b == self.a**2

A = Teste(3)
print(A == 9)
print(A == 3)
print()
print(A.a == 9)
print(A.a == 3)
```

```
ôxe, que pergunta da gota!!!
True
ôxe, que pergunta da gota!!!
False

False
True
```

[179]:
```
# Lista com indice começando de 1 e não 0
class Lista(list):
    def __init__(self, lista):
        self.lista = lista

    def __getitem__(self, posicao):
        if type(posicao)==int:
            return list(self.lista)[posicao-1]

        if type(posicao)==slice:
            return list(self.lista)[posicao.start-1:posicao.stop-1:posicao.step]

    def __call__(self, n):
        return self.lista[0: n]
```

[180]:
```
B0 = [1,2,3,4,5]
B1 = Lista([1,2,3,4,5])

print(B0[4], B1[4])
print(B0[1:4:2], B1[1:4:2])
```

```
5 4
[2, 4] [1, 3]
```

[181]:
```
B1(3)
```

[181]:
```
[1, 2, 3]
```

6 pandas

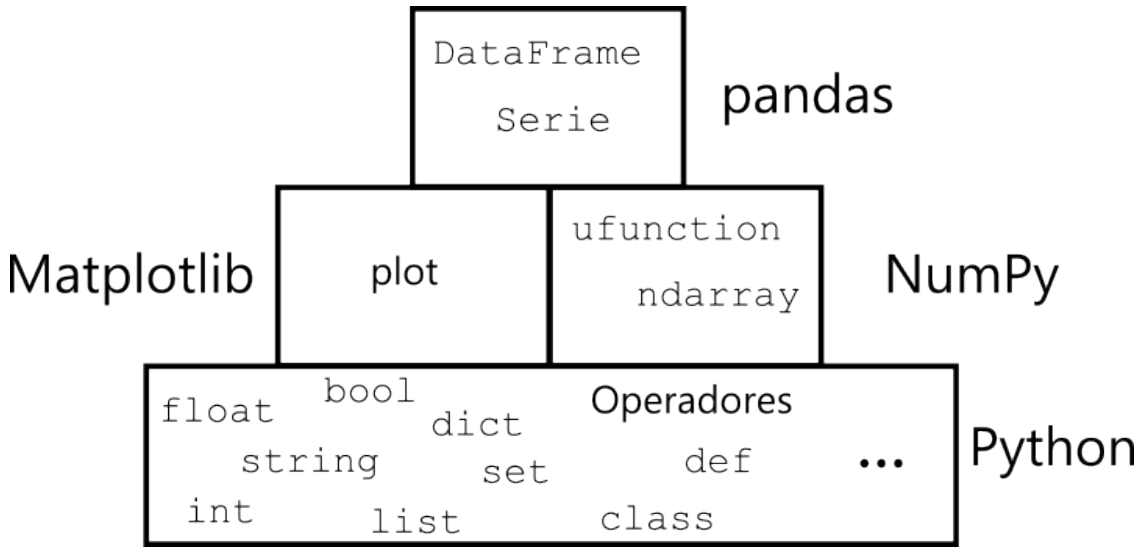
pandas é um pacote Python *open source* rápido, poderoso, flexível e fácil utilizado para aquisição, tratamento e análise de dados. Outras linguagens de programação como R e Julia também contam com pacotes focados nas áreas de engenharia de dados e ciência de dados. Dentro do próprio ecossistema do Python existem outros pacotes como Polars que trabalham nestas áreas, mas o `pandas` está sendo amplamente utilizado com muito material de estudo disponível.

Assim como o `ndarray` é a base do NumPy, o `DataFrame` e `Serie` são os objetos base do pandas. `Serie` é uma lista de dados em apenas uma dimensão, ou seja, uma lista composta por index e apenas uma coluna. O `DataFrame` é uma tabela estruturada formadas de linhas e de colunas.

Os objetos do pandas são construídos com base no Python e NumPy e tem uma forte integração com o Matplotlib para visualização de dados.

Existem pacotes especializados em geração de modelos estatísticos, como o statsmodels, e outros em modelos de aprendizado de máquina, como o scikit-learn, mas este assunto extrapola o objetivo de apresentação de conteúdo deste material.

Vamos fazer um entendimento geral sobre o pacote pandas, depois daremos exemplo de aquisição de dados e ferramentas para tratar e explorar informações de dados.



O pandas roda em ambiente Python, utiliza como base os pacotes Cython e NumPy e tem grande integração como pacote Matplotlib para plotar os dados de um DataFrame ou Serie.

[182]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Limitar números de registros a ser exibido em tela
pd.options.display.max_rows = 15

pd.__version__
```

[182]:
```
'2.2.3'
```

Vamos iniciar criando um DataFrame para entender a estrutura e algumas características básicas.

[183]:
```
datas = pd.date_range('2025-01-01', periods=30)

# DataFrame com dados randômicos com 30 linhas e 4 colunas, indexador de datas e␣
↪colunas de A a D
df = pd.DataFrame(np.random.randn(30, 4), index=datas, columns=['A', 'B', 'C',␣
↪'D'])

df
```

[183]:

	A	B	C	D
2025-01-01	-0.774288	-2.670621	-0.533054	-0.441642
2025-01-02	1.593649	-0.275542	-1.118661	0.265065
2025-01-03	2.008865	-1.566200	0.083626	0.673494
2025-01-04	1.690783	-1.099639	2.101820	0.958331
2025-01-05	0.186816	0.434651	-0.604060	-0.877849
...	...	...	...	...

```
2025-01-26 -0.674493 -0.479123 -1.079783 -0.872899
2025-01-27  0.986006  0.197405  0.813166  0.837139
2025-01-28 -0.623824 -1.263430  1.867787  1.616521
2025-01-29 -0.074684  0.506240  0.259887 -0.750582
2025-01-30  0.359596 -0.054390 -0.596348 -1.148825

[30 rows x 4 columns]
```

O tipo da variável `df` é `DataFrame` do pandas. Cada coluna representa uma `Series` do pandas e os valores guardados na coluna são do tipo `ndarray` no NumPy.

[184]:
```
type(df)
```

[184]:
```
pandas.core.frame.DataFrame
```

Cada coluna do `DataFrame` é um `Serie`.

[185]:
```
df.A
```

[185]:
```
2025-01-01   -0.774288
2025-01-02    1.593649
2025-01-03    2.008865
2025-01-04    1.690783
2025-01-05    0.186816
                ...
2025-01-26   -0.674493
2025-01-27    0.986006
2025-01-28   -0.623824
2025-01-29   -0.074684
2025-01-30    0.359596
Freq: D, Name: A, Length: 30, dtype: float64
```

[186]:
```
type(df.A)
```

[186]:
```
pandas.core.series.Series
```

Uma série pode ser transformada em um `ndarray`do NumPy. Na verdade, `Serie` tem comportamento similar ao tipo `nympy.narray`.

[187]:
```
df.A.values
```

[187]:
```
array([-0.77428779,  1.59364903,  2.00886475,  1.69078252,  0.18681625,
        1.93642341,  1.65580339, -1.533321  ,  0.45719406, -0.49054466,
        0.80287606,  1.45572154,  0.40650429,  1.07006082,  0.18126378,
        1.04617684,  0.69352815,  0.88407559, -1.15097327,  0.43004787,
        1.58002087, -0.12267964, -0.79790663,  0.4086324 , -0.31313024,
       -0.67449319,  0.98600648, -0.62382372, -0.07468411,  0.359596  ])
```

[188]:
```
type(df.A.values)
```

[188]:
```
numpy.ndarray
```

Tanto em `Serie` como em `DataFrame` existe a coluna `index` que é o indexador dos dados.

[189]:
```
df.index
```

[189]:
```
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-03', '2025-01-04',
               '2025-01-05', '2025-01-06', '2025-01-07', '2025-01-08',
               '2025-01-09', '2025-01-10', '2025-01-11', '2025-01-12',
               '2025-01-13', '2025-01-14', '2025-01-15', '2025-01-16',
               '2025-01-17', '2025-01-18', '2025-01-19', '2025-01-20',
               '2025-01-21', '2025-01-22', '2025-01-23', '2025-01-24',
               '2025-01-25', '2025-01-26', '2025-01-27', '2025-01-28',
               '2025-01-29', '2025-01-30'],
              dtype='datetime64[ns]', freq='D')
```

[190]:
```
df.A.index
```

[190]:
```
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-03', '2025-01-04',
               '2025-01-05', '2025-01-06', '2025-01-07', '2025-01-08',
               '2025-01-09', '2025-01-10', '2025-01-11', '2025-01-12',
               '2025-01-13', '2025-01-14', '2025-01-15', '2025-01-16',
               '2025-01-17', '2025-01-18', '2025-01-19', '2025-01-20',
               '2025-01-21', '2025-01-22', '2025-01-23', '2025-01-24',
               '2025-01-25', '2025-01-26', '2025-01-27', '2025-01-28',
               '2025-01-29', '2025-01-30'],
              dtype='datetime64[ns]', freq='D')
```

Exitem métodos do `DataFrame` que ajudam a visualizar e entender a tabela estruturada e seus dados. Usando o método `info` são mostradas as colunas com seus respectivos nomes, valores não nulos e o tipo de dado de cada coluna. Além disso, é mostrado tamanho da tabela com intervalo do `index`e uso da memória pelo `DataFrame`.

[191]:
```
df.info()  # Informações básicas de cada coluna
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 30 entries, 2025-01-01 to 2025-01-30
Freq: D
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   A       30 non-null     float64
 1   B       30 non-null     float64
 2   C       30 non-null     float64
 3   D       30 non-null     float64
dtypes: float64(4)
memory usage: 1.2 KB
```

Ao obter as informações básicas pelo método `info` pode-se coletar um resumo dos dados por meio de medidas estatísticas de cada coluna utilizando o método `describe`, onde é mostrado por padrão o

número de valores diferentes de `NaN` (*Not a Number*, ou valor não informado), média, desvio padrão, valor mínimo, valor máximo e os quartis de 25%, 50% (mediana) e 75%.

[192]:
```
df.describe()
```

[192]:

	A	B	C	D
count	30.000000	30.000000	30.000000	30.000000
mean	0.442607	-0.135272	-0.018772	-0.255083
std	0.961707	1.043976	0.881357	0.956699
min	-1.533321	-2.670621	-1.118661	-2.148273
25%	-0.265518	-0.743481	-0.602132	-0.877552
50%	0.419340	-0.118436	-0.315601	-0.411191
75%	1.064090	0.512610	0.411798	0.457933
max	2.008865	2.001209	2.101820	1.699340

Abaixo estão demonstradas outros métodos que podem ser utilizados para coletar mais detalhes sobre os dados.

[193]:
```
df.head()  # Primeiras 5 linhas
```

[193]:

	A	B	C	D
2025-01-01	-0.774288	-2.670621	-0.533054	-0.441642
2025-01-02	1.593649	-0.275542	-1.118661	0.265065
2025-01-03	2.008865	-1.566200	0.083626	0.673494
2025-01-04	1.690783	-1.099639	2.101820	0.958331
2025-01-05	0.186816	0.434651	-0.604060	-0.877849

[194]:
```
df.tail()  # Últimas 5 linhas
```

[194]:

	A	B	C	D
2025-01-26	-0.674493	-0.479123	-1.079783	-0.872899
2025-01-27	0.986006	0.197405	0.813166	0.837139
2025-01-28	-0.623824	-1.263430	1.867787	1.616521
2025-01-29	-0.074684	0.506240	0.259887	-0.750582
2025-01-30	0.359596	-0.054390	-0.596348	-1.148825

[195]:
```
df.sample(5)  # Amostra aleatória de 5 linhas
```

[195]:

	A	B	C	D
2025-01-07	1.655803	-0.277742	1.142388	0.492002
2025-01-23	-0.797907	0.297707	-0.669955	-1.103536
2025-01-12	1.455722	0.517799	0.202410	0.305044
2025-01-24	0.408632	-0.218479	-0.029919	-0.611840
2025-01-14	1.070061	-0.635959	-0.375746	0.653069

[196]:
```
df.columns  # Nome das colunas
```

[196]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

[197]:
```
df.index  # Lista de indexadores do df
```

[197]:
```
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-03', '2025-01-04',
               '2025-01-05', '2025-01-06', '2025-01-07', '2025-01-08',
               '2025-01-09', '2025-01-10', '2025-01-11', '2025-01-12',
               '2025-01-13', '2025-01-14', '2025-01-15', '2025-01-16',
               '2025-01-17', '2025-01-18', '2025-01-19', '2025-01-20',
               '2025-01-21', '2025-01-22', '2025-01-23', '2025-01-24',
               '2025-01-25', '2025-01-26', '2025-01-27', '2025-01-28',
               '2025-01-29', '2025-01-30'],
              dtype='datetime64[ns]', freq='D')
```

[198]:
```
df.values  # Valores do df em um ndarray
```

[198]:
```
array([[-0.77428779, -2.67062051, -0.53305358, -0.44164211],
       [ 1.59364903, -0.27554177, -1.11866088,  0.26506517],
       [ 2.00886475, -1.56619975,  0.08362644,  0.67349369],
       [ 1.69078252, -1.09963924,  2.10182002,  0.9583309 ],
       [ 0.18681625,  0.43465126, -0.60406031, -0.87784873],
       [ 1.93642341,  0.51473279, -0.34133072, -0.92821   ],
       [ 1.65580339, -0.27774194,  1.14238792,  0.49200197],
       [-1.533321  ,  0.72257433,  1.15211375, -0.45760851],
       [ 0.45719406, -0.74806325,  0.86647319, -2.14827305],
       [-0.49054466, -0.18248296, -0.56371602, -0.38074012],
       [ 0.80287606,  0.71900645, -0.77691899, -0.29758566],
       [ 1.45572154,  0.51779936,  0.20241   ,  0.30504364],
       [ 0.40650429, -1.14388633,  0.44793188, -0.95872378],
       [ 1.07006082, -0.6359586 , -0.37574571,  0.65306918],
       [ 0.18126378,  0.08353884,  0.30339643,  0.91863267],
       [ 1.04617684,  1.91780566,  1.19713413, -2.0988141 ],
       [ 0.69352815,  2.00120881, -0.56348953, -0.87666168],
       [ 0.88407559, -0.72973362, -0.43129592,  0.35572567],
       [-1.15097327,  0.89330651, -1.06654747, -0.99353746],
       [ 0.43004787,  0.48747682, -1.10290817,  1.69934046],
       [ 1.58002087, -2.06288865, -0.28987067, -0.84392279],
       [-0.12267964, -0.79811998,  0.01009376, -0.28310983],
       [-0.79790663,  0.29770698, -0.66995492, -1.10353638],
       [ 0.4086324 , -0.21847933, -0.0299194 , -0.61183953],
       [-0.31313024,  0.85467833, -0.86778533, -0.35250284],
       [-0.67449319, -0.47912326, -1.07978276, -0.8728989 ],
       [ 0.98600648,  0.19740521,  0.81316636,  0.83713894],
       [-0.62382372, -1.26343015,  1.86778713,  1.61652124],
       [-0.07468411,  0.50623969,  0.25988748, -0.75058219],
       [ 0.359596  , -0.05438996, -0.59634837, -1.1488251 ]])
```

Vários métodos estatístico podem ser utilizados, tanto para as colunas como para cada linha. Para definição se a função será aplicada na linha ou na coluna o atributo `axis` é definido. Se `axis=0` a função será aplicada por coluna, ou seja, são processadas todas as linha de cada coluna. Se `axis=1`

a função será aplicada por linha, ou seja, são processadas todas as colunas de cada Linha. Quando não se informa o valor `axis` o valor padrão é zero.

[199]:
```
df.mean()  # Média das linha mostrada por coluna
```

[199]:
```
A    0.442607
B   -0.135272
C   -0.018772
D   -0.255083
dtype: float64
```

[200]:
```
df.mean(axis=1)  # Média das colunas por linha
```

[200]:
```
2025-01-01   -1.104901
2025-01-02    0.116128
2025-01-03    0.299946
2025-01-04    0.912824
2025-01-05   -0.215110
                ...
2025-01-26   -0.776575
2025-01-27    0.708429
2025-01-28    0.399264
2025-01-29   -0.014785
2025-01-30   -0.359992
Freq: D, Length: 30, dtype: float64
```

[201]:
```
df.mean().mean() #  Média das médias de cada coluna
```

[201]:
```
np.float64(0.008369767233884245)
```

Uma coluna pode ser criada de forma simples, com base em informação de outras colunas.

[202]:
```
df = df.assign(E=df.mean(1)) # Criação de nova coluna E com valores médios
```

[203]:
```
# Outra forma de criar coluna em um  colocando em posição específica
df.insert(2,'F', np.sign(df['E'])) # Nova coluna na terceira posição (contagem␣
 ↪começa de 0)
```

[204]:
```
df.head(10) # Listar as primeiras 10 linhas
```

[204]:

	A	B	F	C	D	E
2025-01-01	-0.774288	-2.670621	-1.0	-0.533054	-0.441642	-1.104901
2025-01-02	1.593649	-0.275542	1.0	-1.118661	0.265065	0.116128
2025-01-03	2.008865	-1.566200	1.0	0.083626	0.673494	0.299946
2025-01-04	1.690783	-1.099639	1.0	2.101820	0.958331	0.912824
2025-01-05	0.186816	0.434651	-1.0	-0.604060	-0.877849	-0.215110
2025-01-06	1.936423	0.514733	1.0	-0.341331	-0.928210	0.295404
2025-01-07	1.655803	-0.277742	1.0	1.142388	0.492002	0.753113
2025-01-08	-1.533321	0.722574	-1.0	1.152114	-0.457609	-0.029060

```
2025-01-09  0.457194 -0.748063 -1.0  0.866473 -2.148273 -0.393167
2025-01-10 -0.490545 -0.182483 -1.0 -0.563716 -0.380740 -0.404371
```

[205]:
```
df['G'] = df['A']*df['B'] + df['C']  # calcula nova coluna G = A*B + C
```

[206]:
```
df.sample(6)
```

[206]:
```
                   A         B    F         C         D         E         G
2025-01-20  0.430048  0.487477  1.0 -1.102908  1.699340  0.378489 -0.893270
2025-01-23 -0.797907  0.297707 -1.0 -0.669955 -1.103536 -0.568423 -0.907497
2025-01-11  0.802876  0.719006  1.0 -0.776919 -0.297586  0.111844 -0.199646
2025-01-02  1.593649 -0.275542  1.0 -1.118661  0.265065  0.116128 -1.557778
2025-01-22 -0.122680 -0.798120 -1.0  0.010094 -0.283110 -0.298454  0.108007
2025-01-08 -1.533321  0.722574 -1.0  1.152114 -0.457609 -0.029060  0.044175
```

[207]:
```
# Trocar valor -1 por Neg e 1 por Pos
df['F'] = df['F'].replace({-1: 'Neg', 1: 'Pos'})
```

[208]:
```
df.head()
```

[208]:
```
                   A         B    F         C         D         E         G
2025-01-01 -0.774288 -2.670621  Neg -0.533054 -0.441642 -1.104901  1.534775
2025-01-02  1.593649 -0.275542  Pos -1.118661  0.265065  0.116128 -1.557778
2025-01-03  2.008865 -1.566200  Pos  0.083626  0.673494  0.299946 -3.062657
2025-01-04  1.690783 -1.099639  Pos  2.101820  0.958331  0.912824  0.242569
2025-01-05  0.186816  0.434651  Neg -0.604060 -0.877849 -0.215110 -0.522860
```

[209]:
```
# Seleção por index e nome de colunas
# df.loc[<intervalo de index>, <nome de colunas>]
df.loc['2023-01-05':'2023-01-15':, ['B', 'E']]
```

[209]:
```
Empty DataFrame
Columns: [B, E]
Index: []
```

[210]:
```
# Seleção por índice
# df.iloc[<intervalo de linhas>, <colunas>]
df.iloc[4:15, [1,3]]
```

[210]:
```
                   B         C
2025-01-05  0.434651 -0.604060
2025-01-06  0.514733 -0.341331
2025-01-07 -0.277742  1.142388
2025-01-08  0.722574  1.152114
2025-01-09 -0.748063  0.866473
2025-01-10 -0.182483 -0.563716
2025-01-11  0.719006 -0.776919
2025-01-12  0.517799  0.202410
```

```
2025-01-13 -1.143886  0.447932
2025-01-14 -0.635959 -0.375746
2025-01-15  0.083539  0.303396
```

[211]:
```
# df.loc[<critério de seleção de linha>, <nome da colunas>] = <novo valor>
df.loc[df['C']<0, 'B'] = 0
```

[212]:
```
df.loc[:,['B','C']].head(8)
```

[212]:
```
                   B         C
2025-01-01  0.000000 -0.533054
2025-01-02  0.000000 -1.118661
2025-01-03 -1.566200  0.083626
2025-01-04 -1.099639  2.101820
2025-01-05  0.000000 -0.604060
2025-01-06  0.000000 -0.341331
2025-01-07 -0.277742  1.142388
2025-01-08  0.722574  1.152114
```

Uma funcionalidade muito útil é a função `pandas.DataFrame.groupby` que realiza agrupamento de acordo com valores de colunas informadas e aplica determinada função.

[213]:
```
df.groupby('F').mean()
```

[213]:
```
            A         B         C         D         E         G
F
Neg -0.168884 -0.097417 -0.237636 -0.815039 -0.382871 -0.517283
Pos  1.054097 -0.099364  0.200092  0.304873  0.399611  0.180054
```

[214]:
```
df.F.value_counts()
```

[214]:
```
F
Neg    15
Pos    15
Name: count, dtype: int64
```

[215]:
```
# Resultado em valor normalizado (entre 0 e 1)
df.F.value_counts(normalize=True)
```

[215]:
```
F
Neg    0.5
Pos    0.5
Name: proportion, dtype: float64
```

[216]:
```
# Valores divididos em 3 intervalos e contar para cada intervalo
df.A.value_counts(bins=3)
```

```
[216]: (-0.353, 0.828]                  12
       (0.828, 2.009]                   11
       (-1.5379999999999998, -0.353]     7
       Name: count, dtype: int64
```

6.1 Aquisição de dados

O `pandas` conta com vários métodos de leitura de dados para criação de `DataFrame`. Abaixo temos listados os métodos, breve descrição e links para documentação oficial.

Método	Fonte do dado
read_csv	Arquivo CSV
read_excel	Microsoft Excel
read_fwf	Colunas de largura fixa
read_table	Arquivo com delimitador de coluna em geral
read_html	Tabela HTML
read_json	Arquivo JSON
read_xml	Documento em formato XML
read_clipboard	MS Windows Clipboard
read_gbq	Google BigQuery
read_hdf	Arquivo HDF5
read_pickle	Arquivo Pickle
read_sas	Arquivo SAS em formato XPORT ou SAS7BDAT
read_sql	Base de Dados SQL
read_sql_query	Resultado de query string
read_sql_table	Tabela SQL
read_stata	Arquivo Stata
read_orc	Objeto ORC
read_feather	Arquivos Feather
read_parquet	Objeto Parquet
read_spss	Arquivo SPSS

Da mesma forma, existem vários métodos para exportação de dados:

to_clipboard	MS Windows Clipboard
to_csv	Arquivo csv
to_dict	Dicionário Python
to_excel	Arquivo MS Excel
to_feather	Formato Feather
to_gbq	Tabela Google BigQuery
to_hdf	Arquivo HDF5
to_html	Arquivo HTML
to_json	String JSON
to_latex	Tabela LaTeX
to_markdown	Formato Markdown
to_numpy	`dnarray` (NumPy)

to_records	dnarray (NumPy) com mais opções
to_orc	Objeto ORC
to_parquet	Arquivo binário em formato Parquet
to_period	Formato PeriodIndex
to_pickle	Arquivo Pickle
to_sql	Base de dados SQL
to_stata	Arquivo dta formato Stata
to_string	Tabular console-friendly
to_timestamp	DatetimeArray
to_xarray	Objeto xarray
to_xml	Arquivo XML

[217]:
```
df_m5 = pd.read_csv('FATOR_CAPACIDADE-2_2022_05.csv', sep=';')
df_m5.head()
```

[217]:
```
  id_subsistema nom_subsistema id_estado nom_estado  \
0             N          Norte        MA   MARANHAO
1            NE       Nordeste        BA      BAHIA
2            NE       Nordeste        BA      BAHIA
3            NE       Nordeste        BA      BAHIA
4            NE       Nordeste        BA      BAHIA

            nom_pontoconexao nom_localizacao  val_latitude  val_longitude  \
0            MIRANDA II500kVA        Interior     -2.727222     -42.596389
1      PINDAI II - 230 kV (A)        Interior    -14.353933     -42.575842
2     IGAPORA II - 230 kV (B)        Interior    -14.102794     -42.609369
3   U.SOBRADINHO - 500 kV (A)        Interior     -9.751812     -41.006198
4  MORRO CHAPEU2 - 230 kV (A)        Interior    -10.970000     -41.228000

  nom_modalidadeoperacao nom_tipousina   nom_usina_conjunto  \
0     Conjunto de Usinas        Eólica  Conj. Paulino Neves
1     Conjunto de Usinas        Eólica         Conj. Abil I
2     Conjunto de Usinas        Eólica         Conj. Araçás
3     Conjunto de Usinas        Eólica        Conj. Arizona
4     Conjunto de Usinas        Eólica      Conj. Babilônia

          din_instante  val_geracao  val_capacidadeinstalada  \
0  2022-05-01 00:00:00        1.234                   426.00
1  2022-05-01 00:00:00       61.016                    90.00
2  2022-05-01 00:00:00      126.185                   167.70
3  2022-05-01 00:00:00       69.273                   124.74
4  2022-05-01 00:00:00      116.351                   136.50

   val_fatorcapacidade
0             0.002897
1             0.677956
```

```
2             0.752445
3             0.555339
4             0.852388
```

[218]:
```
df_m5.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 120648 entries, 0 to 120647
Data columns (total 15 columns):
 #   Column                   Non-Null Count   Dtype
---  ------                   --------------   -----
 0   id_subsistema            120648 non-null  object
 1   nom_subsistema           120648 non-null  object
 2   id_estado                120648 non-null  object
 3   nom_estado               120648 non-null  object
 4   nom_pontoconexao         119904 non-null  object
 5   nom_localizacao          111720 non-null  object
 6   val_latitude             119904 non-null  float64
 7   val_longitude            119904 non-null  float64
 8   nom_modalidadeoperacao   120648 non-null  object
 9   nom_tipousina            120648 non-null  object
 10  nom_usina_conjunto       120648 non-null  object
 11  din_instante             120648 non-null  object
 12  val_geracao              120648 non-null  float64
 13  val_capacidadeinstalada  120648 non-null  float64
 14  val_fatorcapacidade      120648 non-null  float64
dtypes: float64(5), object(10)
memory usage: 13.8+ MB
```

[219]:
```
df_m6 = pd.read_excel('FATOR_CAPACIDADE-2_2022_06.xlsx')
df_m6.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 112536 entries, 0 to 112535
Data columns (total 15 columns):
 #   Column                   Non-Null Count   Dtype
---  ------                   --------------   -----
 0   id_subsistema            112536 non-null  object
 1   nom_subsistema           112536 non-null  object
 2   id_estado                112536 non-null  object
 3   nom_estado               112536 non-null  object
 4   nom_pontoconexao         111864 non-null  object
 5   nom_localizacao          104472 non-null  object
 6   val_latitude             111744 non-null  float64
 7   val_longitude            111744 non-null  float64
 8   nom_modalidadeoperacao   112536 non-null  object
 9   nom_tipousina            112536 non-null  object
 10  nom_usina_conjunto       112536 non-null  object
```

```
 11  din_instante              112536 non-null  datetime64[ns]
 12  val_geracao               112536 non-null  float64
 13  val_capacidadeinstalada   112536 non-null  float64
 14  val_fatorcapacidade       112536 non-null  float64
dtypes: datetime64[ns](1), float64(5), object(9)
memory usage: 12.9+ MB
```

É comum que se tenha mais de uma fonte de dados e que seja necessário concatenar tabelas em um único `Data Frame`. O método `pandas.concat` pode ser usado para esta tarefa.

[220]:
```
# Concatenar Data Frames
dfc = pd.concat([df_m5, df_m6])

dfc.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 233184 entries, 0 to 112535
Data columns (total 15 columns):
 #   Column                    Non-Null Count   Dtype
---  ------                    --------------   -----
 0   id_subsistema             233184 non-null  object
 1   nom_subsistema            233184 non-null  object
 2   id_estado                 233184 non-null  object
 3   nom_estado                233184 non-null  object
 4   nom_pontoconexao          231768 non-null  object
 5   nom_localizacao           216192 non-null  object
 6   val_latitude              231648 non-null  float64
 7   val_longitude             231648 non-null  float64
 8   nom_modalidadeoperacao    233184 non-null  object
 9   nom_tipousina             233184 non-null  object
 10  nom_usina_conjunto        233184 non-null  object
 11  din_instante              233184 non-null  object
 12  val_geracao               233184 non-null  float64
 13  val_capacidadeinstalada   233184 non-null  float64
 14  val_fatorcapacidade       233184 non-null  float64
dtypes: float64(5), object(10)
memory usage: 28.5+ MB
```

[221]:
```
dfc.memory_usage(deep=True)  # Uso de memória de cada coluna
```

[221]:
```
Index                     1865472
id_subsistema            11873976
nom_subsistema           13339944
id_estado                11892384
nom_estado               13880088
                           ...
nom_usina_conjunto       16593960
din_instante             22608672
```

```
val_geracao                 1865472
val_capacidadeinstalada     1865472
val_fatorcapacidade         1865472
Length: 16, dtype: int64
```

O limite para quantidade dados a ser tratada no pandas é definido pela memória RAM disponível. O tipo de dado da variáveis, `dtype`, é um aspecto importante para otimização de dados. Na importação de dados o pandas define de forma automática o tipo de dados de cada coluna, caso não sea declarado. A definição de `dtype` para cada coluna reduz a memória RAM utilizada pela DataFrame. No caso do exemplo abaixo, temos uma redução de utilização de memória RAM para menos da metade.

[222]:
```
# Redefinir tipo de cada coluna com intuito de diminuir memória
# Esta definição poderia ser feita no momento da criação do df
dic_dtype = {'id_subsistema': 'category',
             'nom_subsistema': 'category',
             'id_estado': 'category',
             'nom_estado': 'category',
             'nom_pontoconexao': 'category',
             'nom_localizacao': 'category',
             'val_latitude': np.float32,
             'val_longitude': np.float32,
             'nom_modalidadeoperacao': str,
             'nom_tipousina': 'category',
             'nom_usina_conjunto': str,
             'din_instante':'datetime64[ns]',
             'val_geracao': np.float32,
             'val_capacidadeinstalada': np.float32,
             'val_fatorcapacidade': np.float32}
dfc1 = dfc.astype(dic_dtype)  # Redefinindo datatype das colunas
dfc1.index = dfc1.index.astype(np.int32)  # Redefinindo datatype do index
```

[223]:
```
dfc1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 233184 entries, 0 to 112535
Data columns (total 15 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   id_subsistema           233184 non-null  category
 1   nom_subsistema          233184 non-null  category
 2   id_estado               233184 non-null  category
 3   nom_estado              233184 non-null  category
 4   nom_pontoconexao        231768 non-null  category
 5   nom_localizacao         216192 non-null  category
 6   val_latitude            231648 non-null  float32
 7   val_longitude           231648 non-null  float32
 8   nom_modalidadeoperacao  233184 non-null  object
```

```
 9   nom_tipousina            233184 non-null  category
 10  nom_usina_conjunto       233184 non-null  object
 11  din_instante             233184 non-null  datetime64[ns]
 12  val_geracao              233184 non-null  float32
 13  val_capacidadeinstalada  233184 non-null  float32
 14  val_fatorcapacidade      233184 non-null  float32
dtypes: category(7), datetime64[ns](1), float32(5), object(2)
memory usage: 12.2+ MB
```

[224]:
```
print('Memória dfc1 / dfc: ', end='')
print(f'{dfc1.memory_usage().sum() / dfc.memory_usage().sum():.2%}')
print()
print(dfc1.memory_usage(deep=True)  / dfc.memory_usage(deep=True))
```

```
Memória dfc1 / dfc: 42.98%

Index                      0.500000
id_subsistema              0.019670
nom_subsistema             0.017510
id_estado                  0.019680
nom_estado                 0.016868
                               ...
nom_usina_conjunto         1.000000
din_instante               0.082511
val_geracao                0.500000
val_capacidadeinstalada    0.500000
val_fatorcapacidade        0.500000
Length: 16, dtype: float64
```

[225]:
```
df_ansi = pd.read_html('http://engelco.com.br/tabela-ansi/')[0]
df_ansi
```

[225]:

	0	1
0	NR	DENOMINAÇÃO
1	1	Elemento Principal
2	2	Relé de partida ou fechamento temporizado
3	3	Relé de verificação ou interbloqueio
4	4	Contator principal
..	...	...
112	RIO	Dispositivo Remoto de Inputs/Outputs
113	RTU	Unidade de terminal remoto / Concentrador de D...
114	SER	Sistema de armazenamento de eventos
115	TCM	Esquema de monitoramento de Trip
116	SOTF	Fechamento sob falta

[117 rows x 2 columns]

Colocamos o `[0]` no fim da linha para trazer a primeira tabela. O retorno da função `pd.read_html` é uma lista do Python com todas as tabelas encontradas em uma página html, sendo `0` a primeira

tabela encontrada, 1 a segunda tabela, e assim por diante.

[226]:
```
df_ff = pd.read_fwf('faithful.dat')
df_ff.head()
```

[226]:

	ID	eruptions	waiting
0	1	3.600	79
1	2	1.800	54
2	3	3.333	74
3	4	2.283	62
4	5	4.533	85

6.2 Exemplo de Análise Exploratória de Dados

[227]:
```
cod_bcb = 433 # Código para coleta do IPCA no API do Banco Central
url = f'https://api.bcb.gov.br/dados/serie/bcdata.sgs.{cod_bcb}/dados?
 ↪formato=json'
df_ipca = pd.read_json(url)
df_ipca.tail(10)
```

[227]:

	data	valor
530	01/04/2024	0.38
531	01/05/2024	0.46
532	01/06/2024	0.21
533	01/07/2024	0.38
534	01/08/2024	-0.02
535	01/09/2024	0.44
536	01/10/2024	0.56
537	01/11/2024	0.39
538	01/12/2024	0.52
539	01/01/2025	0.16

[228]:
```
df_ipca.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 540 entries, 0 to 539
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   data    540 non-null    object 
 1   valor   540 non-null    float64
dtypes: float64(1), object(1)
memory usage: 8.6+ KB
```

[229]:
```
# Transformar coluna data em formaro datetime
df_ipca['data'] = pd.to_datetime(df_ipca['data'], dayfirst=True)

# Defino coluna data como indice do data frame
```

```
df_ipca.set_index('data', inplace=True)

# Trocar nome da coluna "valor" para "ipca"
df_ipca.columns = ['ipca']

# Tambem se pode renomear usando função rename
df_ipca.tail()
```

[229]:

data	ipca
2024-09-01	0.44
2024-10-01	0.56
2024-11-01	0.39
2024-12-01	0.52
2025-01-01	0.16

[230]:
```
df_ipca.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 540 entries, 1980-02-01 to 2025-01-01
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   ipca    540 non-null    float64
dtypes: float64(1)
memory usage: 8.4 KB
```

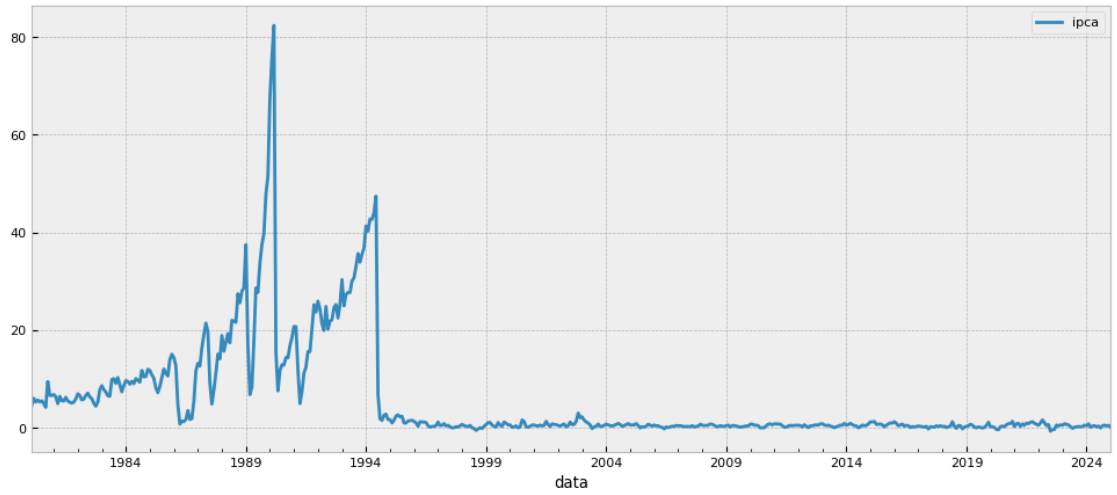
[231]:
```
df_ipca.loc['2022-01-01':'2022-12-01'].describe()
```

[231]:

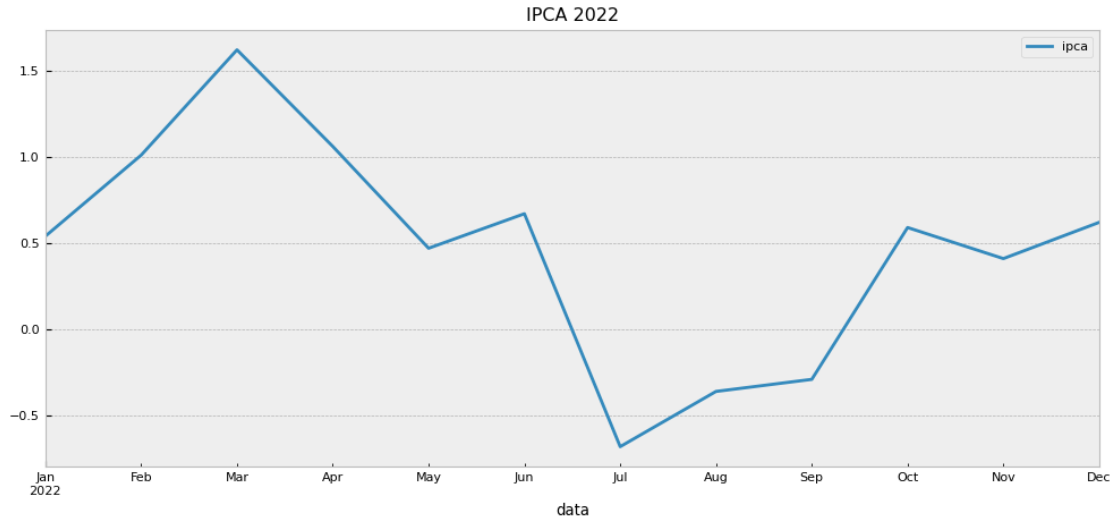
	ipca
count	12.000000
mean	0.471667
std	0.649235
min	-0.680000
25%	0.235000
50%	0.565000
75%	0.755000
max	1.620000

O pandas conta com uma integração com a biblioteca matplotlib, conseguindo usar métodos de plotagem direto do DataFrame. Abaixo estão alguns exemplos de plotagem de gráfico de linhas, barras, histograma, diagrama de caixa (boxplot), gráfico de diferenças e de autocorrelação.

[232]:
```
# Gráfico histórico do IPCA
df_ipca.plot(figsize=(12,5))
plt.show()
```



[233]:
```
# Gráfico IPCA 2022
df_ipca.loc['2022-01-01':'2022-12-01'].plot(figsize=(12,5), title='IPCA 2022')
plt.show()
```



[234]:
```
# IPCA entre janeiro de 2023 até último registro
df_ipca.loc['2023-01-01':]
```

[234]:

data	ipca
2023-01-01	0.53
2023-02-01	0.84
2023-03-01	0.71

```
2023-04-01  0.61
2023-05-01  0.23
...          ...
2024-09-01  0.44
2024-10-01  0.56
2024-11-01  0.39
2024-12-01  0.52
2025-01-01  0.16

[25 rows x 1 columns]
```

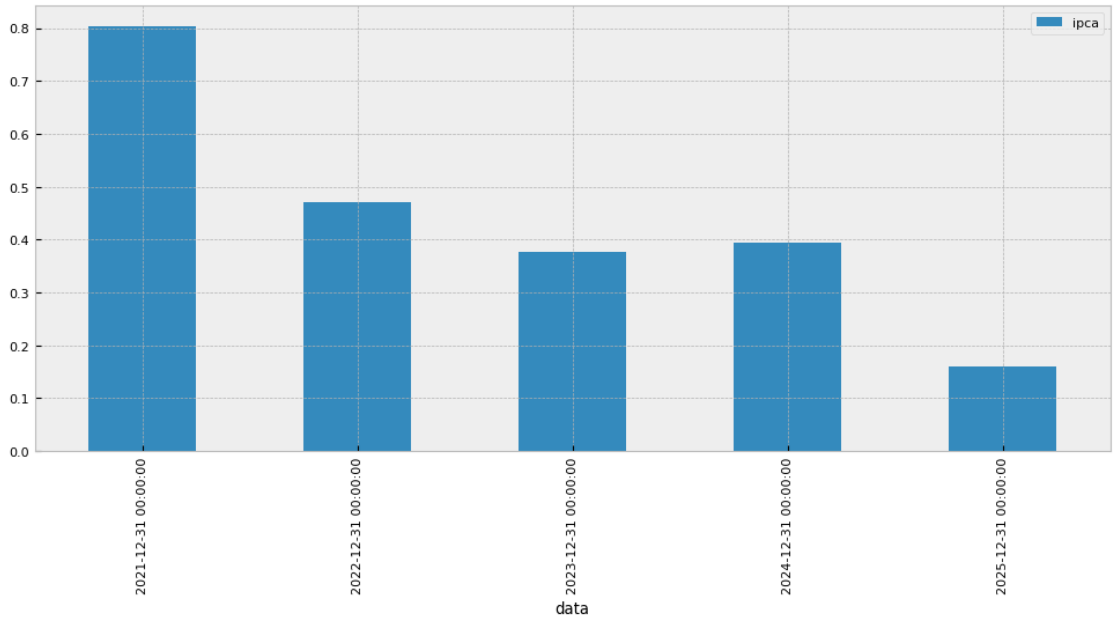
[235]:
```
df_ipca_4a = df_ipca.loc['2021-01-01':]
```

[236]:
```
# Refazer amostra de forma anual usando média do IPCA
df_ipca_4a.resample('YE').mean()
```

[236]:
```
                ipca
data
2021-12-31  0.802500
2022-12-31  0.471667
2023-12-31  0.377500
2024-12-31  0.394167
2025-12-31  0.160000
```

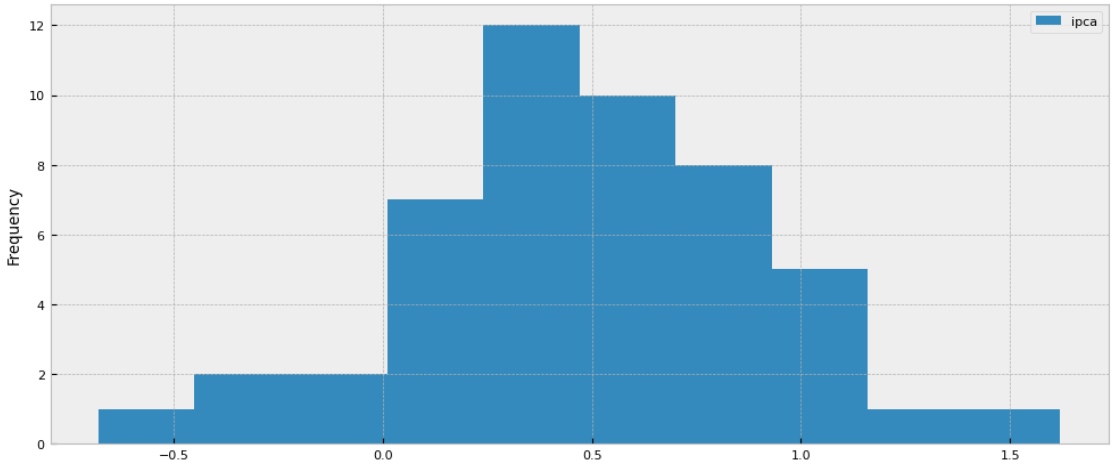
[237]:
```
df_ipca_4a.resample('YE').mean().plot.bar(figsize=(12,5))
plt.plot()
```

[237]:
```
[]
```



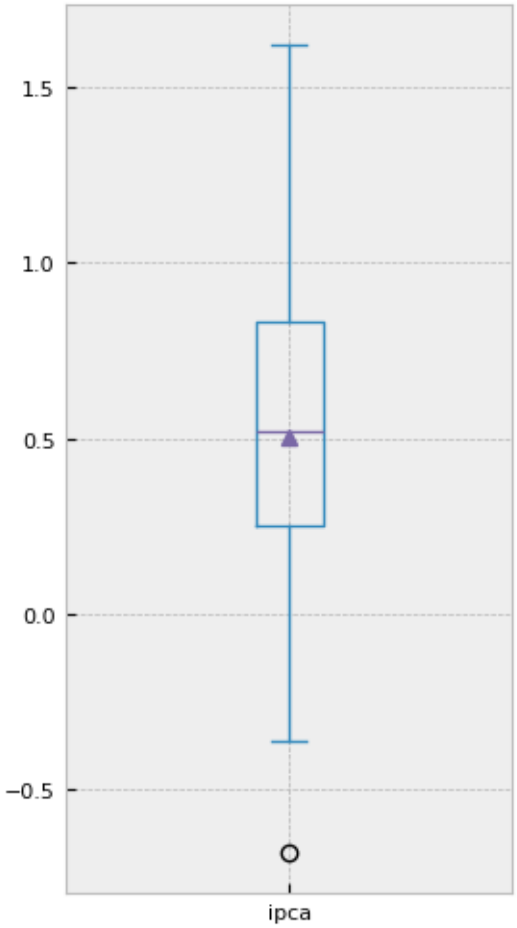
[238]:
```
df_ipca_4a.plot.hist(figsize=(12,5))
plt.plot()
```

[238]: []



[239]:
```
print(df_ipca_4a.describe().T)
df_ipca_4a.plot.box(showmeans=True, figsize=(3,6))
plt.show()
```

```
      count      mean       std   min   25%   50%   75%   max
ipca   49.0  0.504286  0.420758 -0.68  0.25  0.52  0.83  1.62
```



[240]:
```
# Somar unidade a cada índice e calcular inflação acumulada
print('***',2023,'***')
print((df_ipca.loc['2023-01-01':]/100 + 1).apply(np.cumprod))
print()
print('***',2022,'***')
print(((df_ipca.loc['2022-01-01':'2023-01-01']/100 + 1).apply(np.cumprod)))
```

```
*** 2023 ***
                ipca
data
2023-01-01  1.005300
2023-02-01  1.013745
2023-03-01  1.020942
2023-04-01  1.027170
2023-05-01  1.029532
...              ...
2024-09-01  1.080792
2024-10-01  1.086844
```

```
2024-11-01  1.091083
2024-12-01  1.096757
2025-01-01  1.098512

[25 rows x 1 columns]

*** 2022 ***
                ipca
data
2022-01-01  1.005400
2022-02-01  1.015555
2022-03-01  1.032007
2022-04-01  1.042946
2022-05-01  1.047848
2022-06-01  1.054868
2022-07-01  1.047695
2022-08-01  1.043923
2022-09-01  1.040896
2022-10-01  1.047037
2022-11-01  1.051330
2022-12-01  1.057848
2023-01-01  1.063455
```

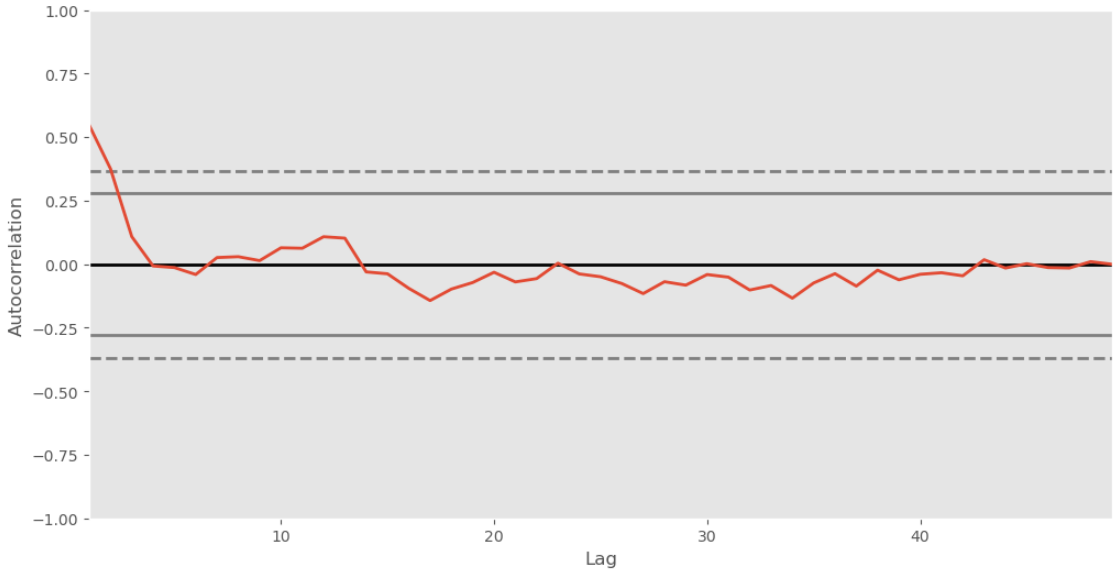
Para traçar um gráfico de médias móveis pode ser usada o método `rolling` definindo o tamanho da janela (no exemplo uma janela de 3 meses) e aplicando o método `mean`.

[241]:
```
plt.style.use('ggplot')
fig, ax = plt.subplots(figsize=(12,6))
df_ipca_4a.plot(ax=ax)
df_ipca_4a.rolling(window=3).mean().plot(ax=ax, style='g:')
ax.legend(['IPCA', 'Média Móvel (3 meses)']);
```

[242]:
```
fig, ax = plt.subplots(figsize=(12,6))
df_ipca_4a.plot(ax=ax)
df_ipca_4a.diff().plot(ax=ax, style='g:')
ax.legend(['IPCA', 'Diferença']);
```

[243]:
```
print(df_ipca_4a.ipca.autocorr())
plt.figure(figsize=(12,6))
pd.plotting.autocorrelation_plot(df_ipca_4a)
plt.show()
```

```
0.5534496560526941
```



[244]:
```
cod_bcb = 189 # IGP-M
url = f'https://api.bcb.gov.br/dados/serie/bcdata.sgs.{cod_bcb}/dados?
 ↪formato=json'
df_igpm = pd.read_json(url)
df_igpm['data'] = pd.to_datetime(df_igpm['data'], dayfirst=True)
df_igpm.set_index('data', inplace=True)
df_igpm.rename(columns={'valor': 'igpm'}, inplace=True)
df_igpm.info()
df_igpm.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 428 entries, 1989-07-01 to 2025-02-01
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   igpm    428 non-null    float64
dtypes: float64(1)
memory usage: 6.7 KB
```

[244]:

data	igpm
2024-10-01	1.52
2024-11-01	1.30
2024-12-01	0.94
2025-01-01	0.27
2025-02-01	1.06

[245]:
```
cod_bcb = 191 # IPC-BR
url = f'https://api.bcb.gov.br/dados/serie/bcdata.sgs.{cod_bcb}/dados?
 ↪formato=json'
df_ipcbr = pd.read_json(url)
df_ipcbr['data'] = pd.to_datetime(df_ipcbr['data'], dayfirst=True)
df_ipcbr.set_index('data', inplace=True)
df_ipcbr.columns=['ipcbr']
df_ipcbr.info()
df_ipcbr.tail()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 421 entries, 1990-02-01 to 2025-02-01
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ipcbr   421 non-null    float64
dtypes: float64(1)
memory usage: 6.6 KB
```

[245]:

data	ipcbr
2024-10-01	0.30
2024-11-01	-0.13
2024-12-01	0.31
2025-01-01	0.02
2025-02-01	1.18

[246]:
```
print(f'IPCA:  {df_ipca.index.min()}',
     f'IGPM:  {df_igpm.index.min()}',
     f'IPCBR: {df_ipcbr.index.min()}', sep='\n')
```

```
IPCA:  1980-02-01 00:00:00
IGPM:  1989-07-01 00:00:00
IPCBR: 1990-02-01 00:00:00
```

O método pandas.DataFrame.merge serve para junção de DataFrames parecido com a funcionalidade que as funções procv/procx fazem no MS Excel.

[247]:
```
# Pegamos como base IPCBR que é o que tem menos dados históricos
# Juntar com dados do IPCA
df_indices = pd.merge(df_ipcbr, df_ipca, how='left',
                      left_index=True, right_index=True)
df_indices.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 421 entries, 1990-02-01 to 2025-02-01
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
```

```
 0   ipcbr   421 non-null    float64
 1   ipca    420 non-null    float64
dtypes: float64(2)
memory usage: 9.9 KB
```

[248]:
```
# Juntar base de IGPM com dados de IPCBR e IPCA
df_indices = df_indices.merge(df_igpm, how='left',
                              left_index=True, right_index=True)
df_indices.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 421 entries, 1990-02-01 to 2025-02-01
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   ipcbr   421 non-null    float64
 1   ipca    420 non-null    float64
 2   igpm    421 non-null    float64
dtypes: float64(3)
memory usage: 13.2 KB
```

[249]:
```
df_4a = df_indices.loc['2021-01-01':]
```

[250]:
```
# Correlação das colunas
df_4a.corr()
```

[250]:

	ipcbr	ipca	igpm
ipcbr	1.000000	0.865134	0.285080
ipca	0.865134	1.000000	0.393113
igpm	0.285080	0.393113	1.000000

[251]:
```
df_4a.describe()
```

[251]:

	ipcbr	ipca	igpm
count	50.00000	49.000000	50.000000
mean	0.43640	0.504286	0.530000
std	0.46658	0.420758	1.140721
min	-1.19000	-0.680000	-1.930000
25%	0.24000	0.250000	-0.040000
50%	0.49500	0.520000	0.555000
75%	0.68500	0.830000	0.927500
max	1.43000	1.620000	4.100000

[252]:
```
df_4a.describe(percentiles=[.10, .25, .5, .75, .9]).T
```

[252]:

	count	mean	std	min	10%	25%	50%	75%	90%	\
ipcbr	50.0	0.436400	0.466580	-1.19	-0.103	0.24	0.495	0.6850	1.008	
ipca	49.0	0.504286	0.420758	-0.68	0.092	0.25	0.520	0.8300	0.970	

```
igpm    50.0  0.530000  1.140721 -1.93 -0.743 -0.04  0.555  0.9275  1.821
```

```
         max
ipcbr  1.43
ipca   1.62
igpm   4.10
```

[253]:
```
df_4a.query('igpm > 3')
```

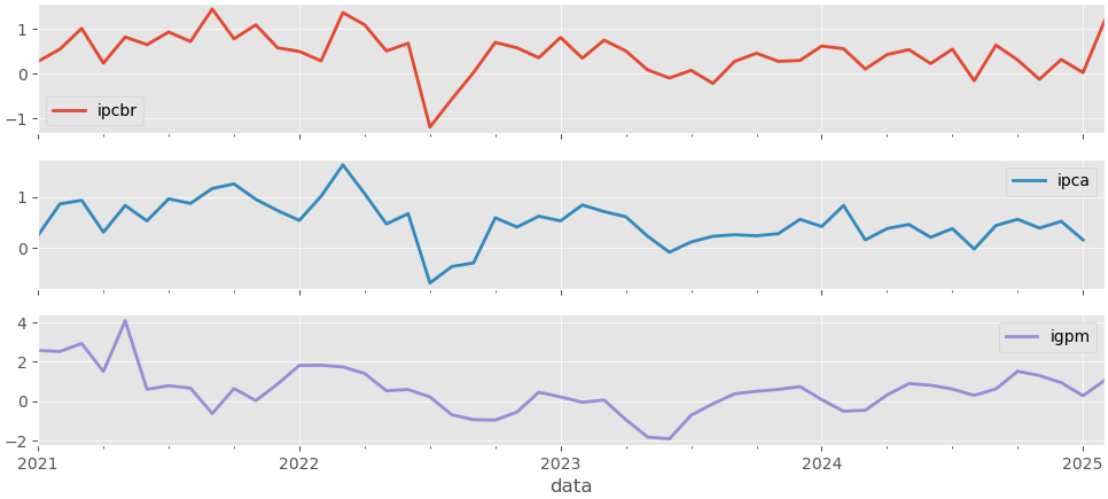
[253]:
```
            ipcbr  ipca  igpm
data
2021-05-01   0.81  0.83   4.1
```

[254]:
```
df_4a.query('igpm > 3 and ipca > 0.85')
```

[254]:
```
Empty DataFrame
Columns: [ipcbr, ipca, igpm]
Index: []
```

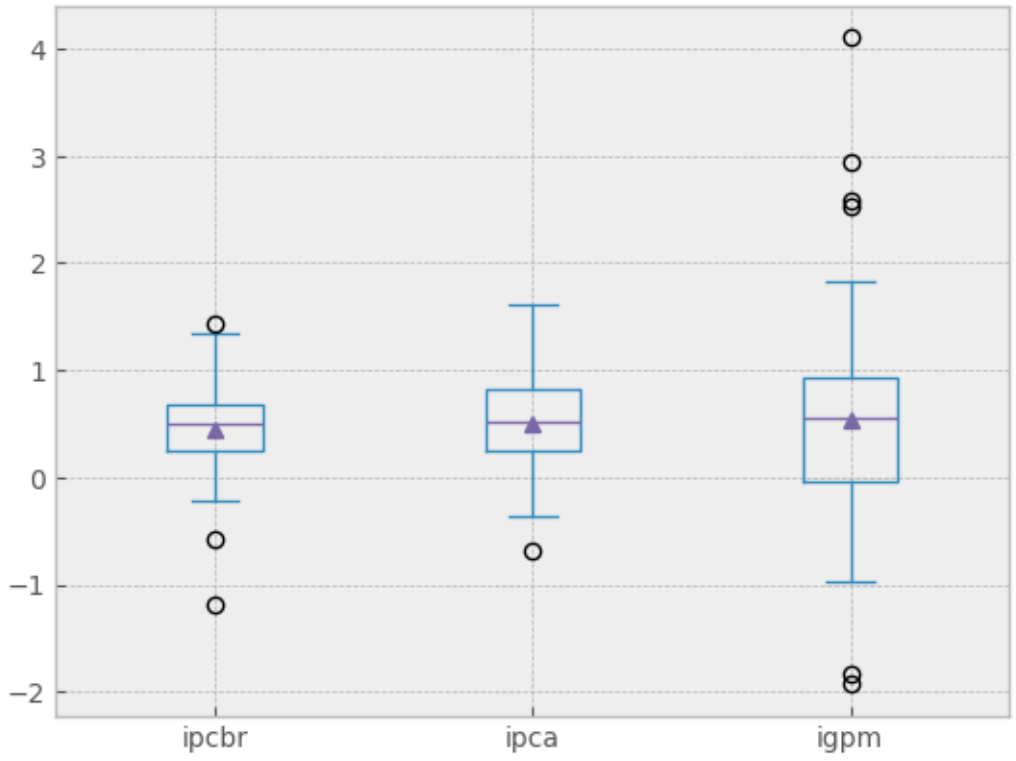
[255]:
```
df_4a.plot(figsize=(12,5))
plt.show()
```

[256]:
```
df_4a.plot(subplots=True, figsize=(12,5))
plt.show()
```

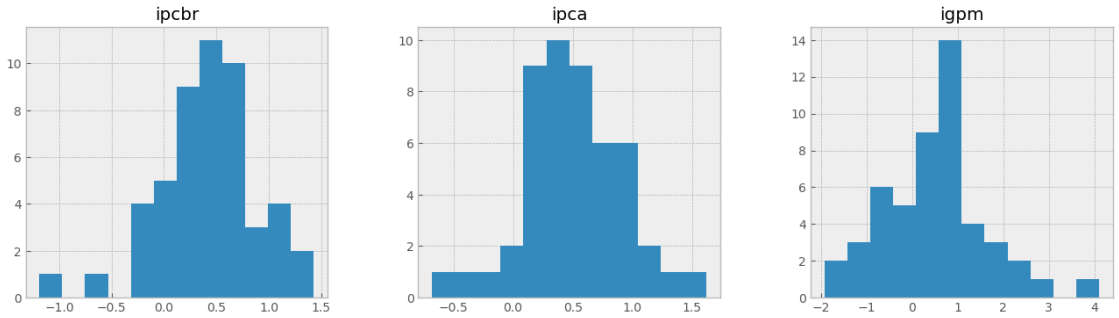


[257]:
```
print(df_4a.describe().T)
plt.style.use('bmh')
df_4a.plot.box(showmeans=True)
plt.show()
```

```
       count      mean       std   min   25%    50%     75%   max
ipcbr   50.0  0.436400  0.466580 -1.19  0.24  0.495  0.6850  1.43
ipca    49.0  0.504286  0.420758 -0.68  0.25  0.520  0.8300  1.62
igpm    50.0  0.530000  1.140721 -1.93 -0.04  0.555  0.9275  4.10
```



[258]:
```
df_4a.hist(bins=12, layout=(1,3), figsize=(16,4))
plt.show()
```



6.3 Seaborn

Seaborn é um pacote de visualização de dados baseado no matplotlib aplicado no Python. É bastante aplicado para visualização em análise estatística de dados.

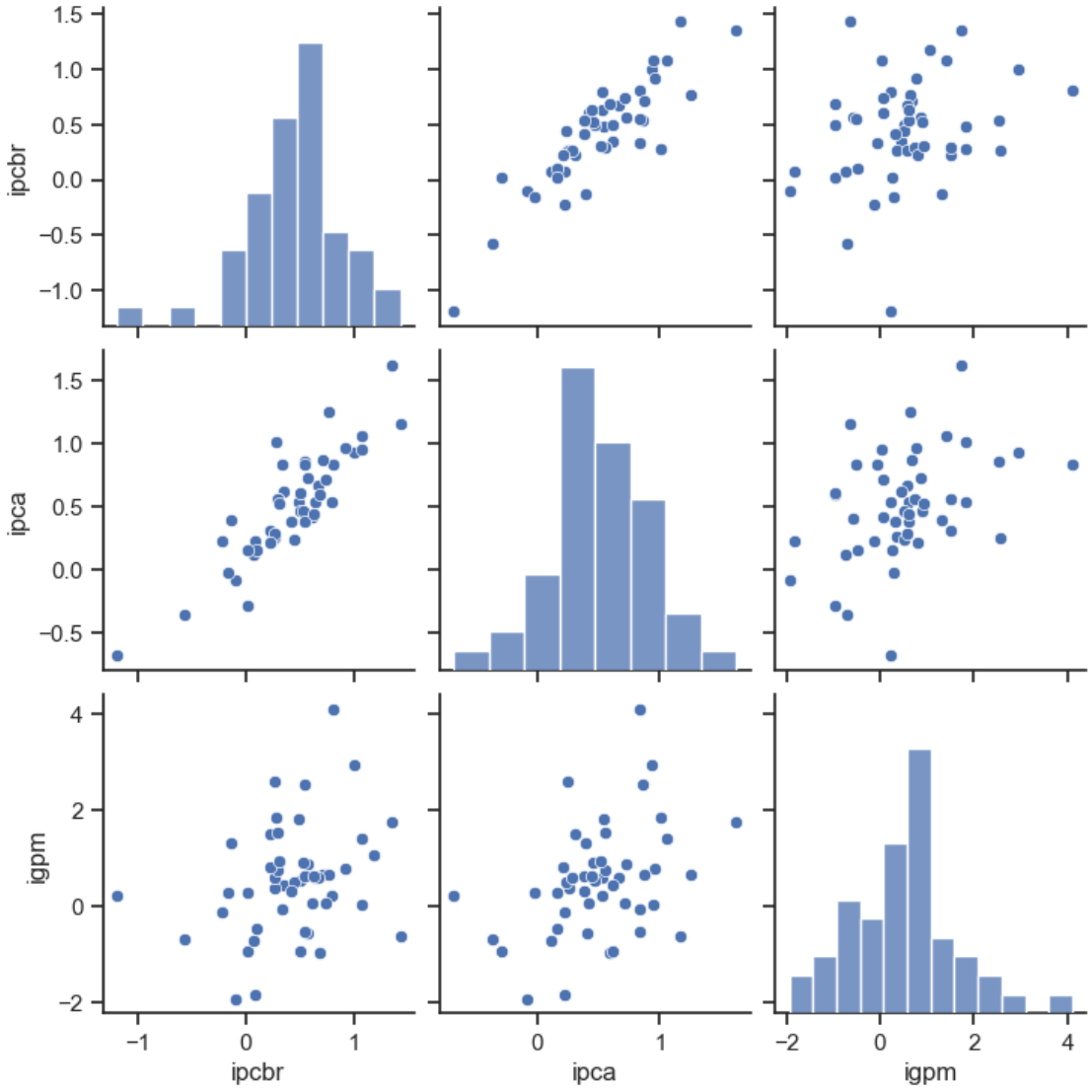
[259]:
```
import seaborn as sns
sns.__version__
```

[259]: '0.13.2'

[260]:
```
print(df_4a.corr())  # Tabela de Correlação
sns.set_theme(style="ticks")
sns.pairplot(df_4a)
```

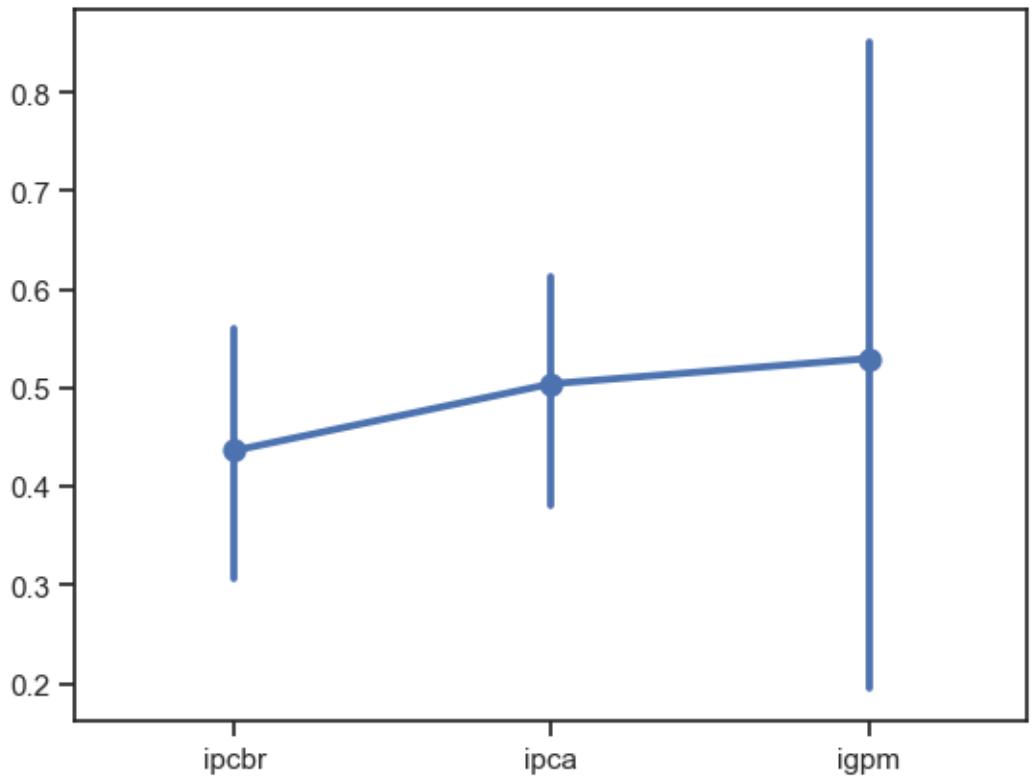
```
          ipcbr      ipca      igpm
ipcbr  1.000000  0.865134  0.285080
ipca   0.865134  1.000000  0.393113
igpm   0.285080  0.393113  1.000000
```

[260]: <seaborn.axisgrid.PairGrid at 0x1c0df3e0cb0>

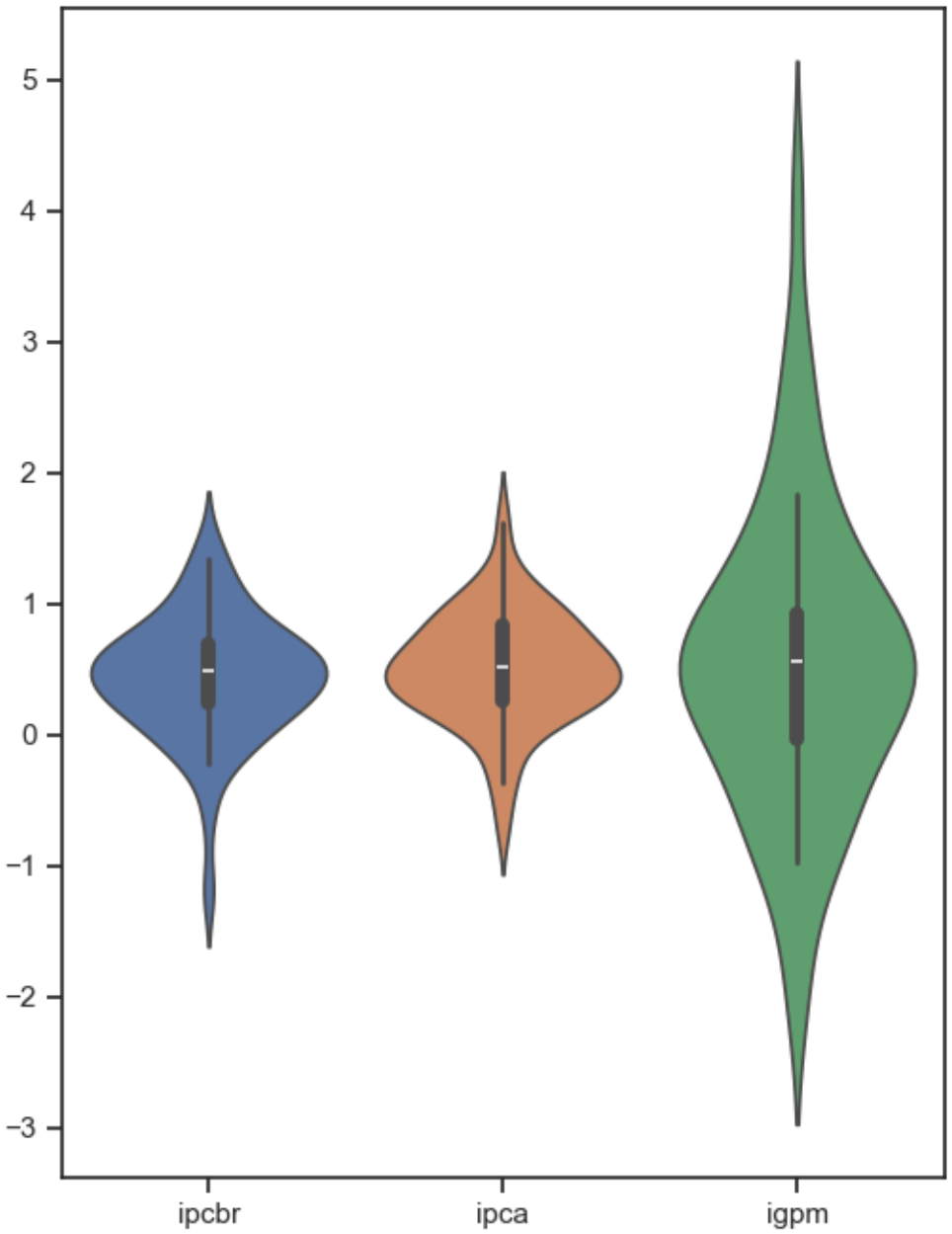


[261]:
```
sns.pointplot(df_4a)
```

[261]: <Axes: >

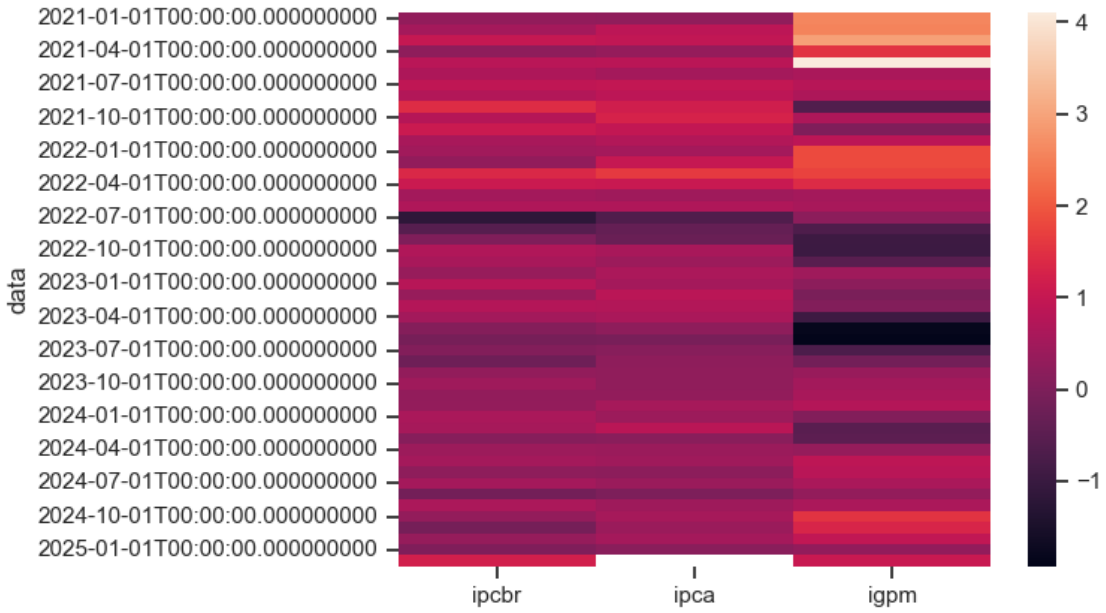


[262]:
```
plt.figure(figsize=(6,8))
sns.violinplot(data=df_4a)
plt.show()
```



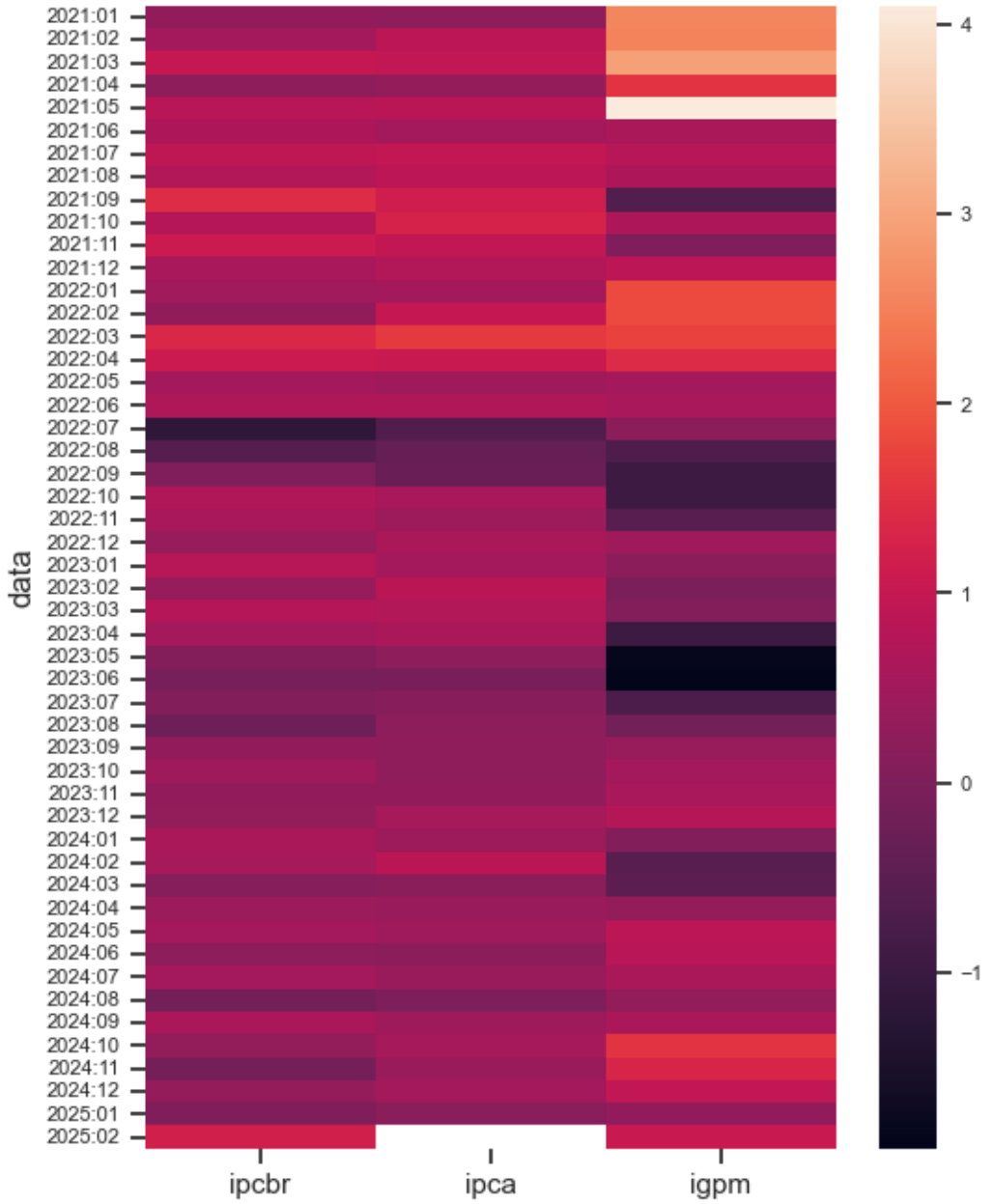
[263]:
```
sns.heatmap(df_4a)
```

[263]:
```
<Axes: ylabel='data'>
```



[264]:
```
plt.figure(figsize=(6, 8))  # Definir tamanho da figura
plt.rc('ytick', labelsize=8)  # Definir tamanho de fonte do eixo y
sns.heatmap(df_4a, yticklabels=df_4a.index.strftime('%Y:%m'))
```

[264]: <Axes: ylabel='data'>



Também se pode colocar outras bibliotecas como a Plotly como padrão para plotar gráficos direto do `pandas`. A biblioteca de gráficos do Plotly para Python cria gráficos interativos de qualidade para publicação.

[265]:
```
pd.options.plotting.backend = "plotly"
```

[266]:
```
df_4a.plot(title='Índices de Inflação', width=1400, height=600)
```

7 Anexos

7.1 Google Colab

- Toda conta Google tem acesso ao ambiente do Colab.
- O Google Colab entende códigos em LaTeX, Markdown e HTML

Gerar PDF de um notebook no Google Colab

```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')

!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('Introdução ao Python.ipynb')
```

7.2 Easter egg no Pytnon - Zen of Python

[267]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```