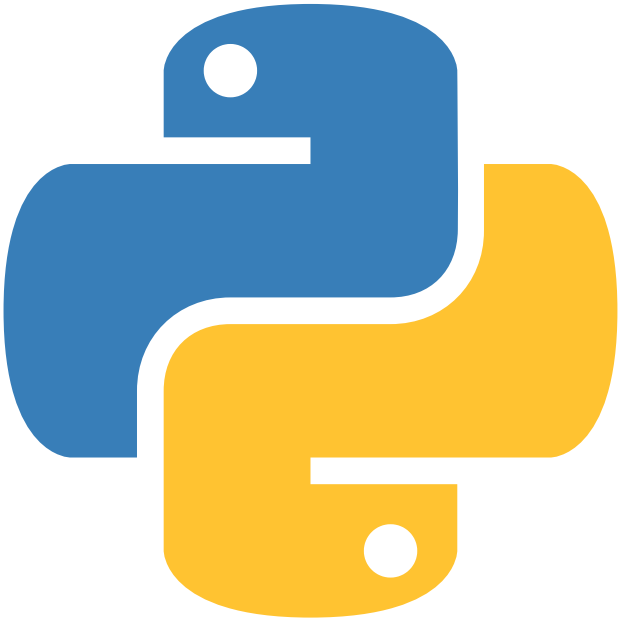
Data Science com Python

Módulo 01 - Introdução ao Python

Nós gostaríamos de te dar as boas vindas ao nosso curso de Data Science com Python.

Este curso foi desenhado para ser uma imersão no mundo de Data Science, unindo o fundamental da teoria com a prática em Python.

Ao longo dos módulos desse curso você vai mergulhar sobre os principais aspectos do mundo de Data Science e como desenvolvê-los usando a programação em Python.



0.Hello World!

Tradicionalmente no ensino de programação, inicia-se com o comando clássico "Hello World!"

In [ ]:
```
# Usando a função "print" para imprimir na tela a mensagem entre aspas

print("Hello World!")
```

```
Hello World!
```

1. Variáveis vs. Objetos

1.1. Tipos de variáveis

In [ ]:
```
3
```

Out[ ]:
```
3
```

In [ ]:
```
a = 3
```

Tipos de variáveis

integer (int): 2

float: 1.45789

string (str): "PETR4"

boolean (bool): True ou False

In [ ]:
```
b = 5.5
```

In [ ]:
```
b
```

Out[ ]:
```
5.5
```

In [ ]:
```
print(b)
```

```
5.5
```

In [ ]:
```
a + b
```

Out[ ]:
```
8.5
```

In [ ]:
```
type(a)
```

Out[ ]:
```
int
```

In [ ]:
```
type(b)
```

Out[ ]:
```
float
```

In [ ]:
```
ativo = 'WEGE3'
```

In [ ]:
```
ativo
```

Out[ ]:
```
'WEGE3'
```

In [ ]:
```
type(ativo)
```

Out[ ]:
```
str
```

In [ ]:
```
a == b
```

Out[ ]:
```
False
```

In [ ]:
```
a = b
```

In [ ]:
```
a
```

Out[ ]:
```
5.5
```

In [ ]:
```
a == b
```

Out[ ]:
```
True
```

In [ ]:
```
c = a == b
```

In [ ]:
```
c
```

Out[ ]:
```
True
```

2. Operadores aritméticos

- Soma +
- Subtração -
- Multiplicação *
- Divisão /
- Divisão // (parte inteira)
- Resto das divisões
- Potência **

In [ ]:
```
d = a + b
```

In [ ]:
```
d
```

Out[ ]:
```
11.0
```

In [ ]:
```
e = d + a + b + d + d
```

In [ ]:
```
e
```

Out[ ]:
```
44.0
```

In [ ]:
```
a-b
```

Out[ ]:
```
0.0
```

In [ ]:
```
e - d
```

Out[ ]:
```
33.0
```

In [ ]:
```
a*b
```

Out[ ]:
```
30.25
```

In [ ]:
```
f = 10
```

In [ ]:
```
e*f
```

Out[ ]:
```
440.0
```

Exemplo

Qual o tamanho da minha posição em PETR4 dados o número de papéis que comprei e a cotação atual?

In [ ]:
```
cotacao = 33.32
papeis = 100
```

In [ ]:
```
patrimonio = cotacao*papeis
```

Qual o preço atual?

In [ ]:
```
patrimonio/papeis
```

Out[ ]:
```
33.32
```

Exemplo com VALE3 - qual o número de papéis que eu posso comprar?

In [ ]:
```
carteira = 4000
```

In [ ]:
```
vale3 = 50.32
```

In [ ]:
```
carteira/vale3
```

Out[ ]:
```
79.4912559618442
```

In [ ]:
```
carteira//vale3
```

Out[ ]:
```
79.0
```

In [ ]:
```
10//4
```

Out[ ]:
```
2
```

Resto da divisão

In [ ]:
```
7%4
```

Out[ ]:
```
3
```

In [ ]:
```
10%3
```

Out[ ]:
```
1
```

Potenciação

In [ ]:
```
2**2
```

Out[ ]:
```
4
```

In [ ]:
```
2**3
```

Out[ ]:
```
8
```

In [ ]:
```
2**10
```

Out[ ]:
```
1024
```

3. Operadores lógicos

3.1. Operadores comparativos

- Igualdade ==
- Diferença !=
- Menor <
- Maior >
- Menor ou igual <=
- Maior ou igual >=

In [ ]:
```
e == f
```

Out[ ]:
```
False
```

In [ ]:
```
e != f
```

Out[ ]:
```
True
```

In [ ]:
```
e
```

Out[ ]:
```
44.0
```

In [ ]:
```
f
```

Out[ ]:
```
10
```

In [ ]:
```
e < f
```

Out[ ]:
```
False
```

In [ ]:
```
e > f
```

Out[ ]:
```
True
```

In [ ]:
```
g = 5
h = 5
```

In [ ]:
```
g >= h
```

Out[ ]:
```
True
```

In [ ]:
```
papel1 = "VALE3"
papel2 = "PETR4"
```

In [ ]:
```
papel1 == papel2
```

Out[ ]:
```
False
```

In [ ]:
```
papel1 != papel2
```

Out[ ]:
```
True
```

3.2. Operadores condicionais

- and
- or

In [ ]:
```
e > f and g >= h
```

Out[ ]: True

In [ ]:
```
e < f
```

Out[ ]: False

In [ ]:
```
e < f or g >= h
```

Out[ ]: True

In [ ]:
```
e < f or g > h
```

Out[ ]: False

In [ ]:
```
e < f and g >= h
```

Out[ ]: False

4. Strings

4.1. Indexação e 'slicing' (fatiamento)

In [ ]:
```
# Suponha que você viu a seguinte manchete:

manchete = 'PETR4 vai aumentar o preço da gasolina'
```

In [ ]:
```
print(manchete)
```

```
PETR4 vai aumentar o preço da gasolina
```

In [ ]:
```
type(manchete)
```

Out[ ]: str

In [ ]:
```
# Comprimento (length) do objeto, nesse caso número de caracteres

len(manchete)
```

Out[ ]: 38

In [ ]:
```
manchete[0]
```

Out[ ]: 'P'

In [ ]:
```
manchete[1]
```

Out[ ]: 'E'

In [ ]:
```
manchete[4]
```

Out[ ]: '4'

In [ ]:
```
manchete[5]
```

Out[ ]: ' '

In [ ]:
```
manchete[0:6]
```

Out[ ]: 'PETR4 '

In [ ]:
```
acao = manchete[0:5]
```

In [ ]:
```
acao
```

Out[ ]: 'PETR4'

In [ ]:
```
# Do caractere 5 até o fim
manchete[5:]
```

Out[ ]: ' vai aumentar o preço da gasolina'

In [ ]:
```
# Do início até o caractere 10 (sem retornar o 10)
manchete[:10]
```

Out[ ]:
```
'PETR4 vai '
```

In [ ]:
```
# Do início até o fim

manchete[:]
```

Out[ ]:
```
'PETR4 vai aumentar o preço da gasolina'
```

In [ ]:
```
# A operação de indexing (indexação) também pode começar 'de trás para frente'. Basta utilizar índices negati
manchete[:-8]
```

Out[ ]:
```
'PETR4 vai aumentar o preço da '
```

In [ ]:
```
manchete[5:-8]
```

Out[ ]:
```
' vai aumentar o preço da '
```

Agora, temos um segundo ':'

Ele indica de quantos em quantos caracteres devemos percorrer o intervalo

In [ ]:
```
# Neste caso, percorremos todo o intervalo, mas indo de -1 em -1 (ou seja, de trás para frente)
manchete[::-1]
```

Out[ ]:
```
'anilosag ad oçerp o ratnemua iav 4RTEP'
```

In [ ]:
```
manchete[::1]
```

Out[ ]:
```
'PETR4 vai aumentar o preço da gasolina'
```

In [ ]:
```
acao[::-1]
```

Out[ ]:
```
'4RTEP'
```

In [ ]:
```
manchete[5::-1]
```

Out[ ]:
```
' 4RTEP'
```

string[a:b:c]

Resumindo o slicing:

Uma forma de definirmos as notações string[a:b:c] seria algo como: Execute um comando de fatiamento da string em intervalos de ‘c’, a partir de ‘a’ (incluso) até ‘b’ (não incluso). Se ‘c’ for negativo, contamos de trás para frente. Quando ‘c’ não está presente no código, o padrão é 1 (i.e., fatiar de 1 em 1 caractere). Se ‘a’ não está presente, então você começa o mais distante possível na direção que você está contando (i.e., se ‘c’ é positivo, começa no primeiro caractere. Se ‘c’ é negativo, começa no último caractere). Se ‘b’ não está presente, então você termina o mais distante possível na direção que está contando (i.e., se ‘c’ é positivo, termina no último caractere. Se ‘c’ é negativo, termina no primeiro caractere).

4.2. Propriedades das strings

- Imutabilidade
- Adição
- Modificação de upper e lower case

In [ ]:
```
manchete
```

Out[ ]:
```
'PETR4 vai aumentar o preço da gasolina'
```

In [ ]:
```
manchete[4:5]
```

Out[ ]:
```
'4'
```

In [ ]:
```
# Output é um erro, já que a string é imutável, ou seja, não consigo substituir o '4' pelo '3' em 'PETR4'
manchete[4:5] = '3'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-05f30390e666> in <module>
```

```
----> 1 manchete[4:5] = '3'

TypeError: 'str' object does not support item assignment
```

In [ ]:
```
# Da mesma forma, não conseguimos substituir mais de um caractere na string
manchete[0:5] = 'VALE3'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-131-d59cd3e532ac> in <module>
----> 1 manchete[0:5] = 'VALE3'

TypeError: 'str' object does not support item assignment
```

In [ ]:
```
manchete + ' por causa da alta do petróleo'
```

Out[ ]:
```
'PETR4 vai aumentar o preço da gasolina por causa da alta do petróleo'
```

In [ ]:
```
complemento = ' por causa da alta do petróleo'
```

In [ ]:
```
frase_completa = manchete + complemento
```

In [ ]:
```
frase_completa
```

Out[ ]:
```
'PETR4 vai aumentar o preço da gasolina por causa da alta do petróleo'
```

In [ ]:
```
# Método lower: modifica todos os caracteres para minúsculos

minusculo = frase_completa.lower()
```

In [ ]:
```
minusculo
```

Out[ ]:
```
'petr4 vai aumentar o preço da gasolina por causa da alta do petróleo'
```

In [ ]:
```
# Método upper: transforma tudo em maiúsculo

maiusculo = minusculo.upper()
```

In [ ]:
```
maiusculo
```

Out[ ]:
```
'PETR4 VAI AUMENTAR O PREÇO DA GASOLINA POR CAUSA DA ALTA DO PETRÓLEO'
```

Separar as palavras dentro de uma string com o método 'split'

In [ ]:
```
frase_separada = frase_completa.split()
```

In [ ]:
```
frase_separada
```

Out[ ]:
```
['PETR4',
 'vai',
 'aumentar',
 'o',
 'preço',
 'da',
 'gasolina',
 'por',
 'causa',
 'da',
 'alta',
 'do',
 'petróleo']
```

In [ ]:
```
frase_completa.split('v')
```

Out[ ]:
```
['PETR4 ', 'ai aumentar o preço da gasolina por causa da alta do petróleo']
```

In [ ]:
```
# Primeira palavra
frase_separada[0]
```

Out[ ]:
```
'PETR4'
```

In [ ]:
```
# Última palavra
frase_separada[-1]
```

Out[ ]:
```
'petróleo'
```

In [ ]:

5. Listas

In [ ]:
```
# Listas são marcadas pelos colchetes e permitem alocação de itens (ou elementos) repetidos
# Estão entre os objetos mais fáceis de manipulação pois aceitam substituições, adições, repetições...

nomes = ['André', 'João', 'José']
```

In [ ]:
```
nomes
```

Out[ ]:
```
['André', 'João', 'José']
```

In [ ]:
```
type(nomes)
```

Out[ ]:
```
list
```

In [ ]:
```
# Permitem misturar strings com números

lista_misturada = ['André', 'João', 'José', 30, 29, 28]
```

In [ ]:
```
lista_misturada
```

Out[ ]:
```
['André', 'João', 'José', 30, 29, 28]
```

In [ ]:
```
type(lista_misturada)
```

Out[ ]:
```
list
```

In [ ]:
```
# Listas seguem o mesmo padrão de indexação e slicing. Cada elemento é separado por ',' (vírgula)

type(lista_misturada[0])
```

Out[ ]:
```
str
```

In [ ]:
```
# Note que o elemento de índice '3' (30) é um número inteiro

type(lista_misturada[3])
```

Out[ ]:
```
int
```

In [ ]:
```
# Conseguimos criar novas strings resultantes de operações com outras strings, inclusive procedentes de lista

'Os aprovados foram ' + nomes[0] + ' e ' + nomes[2]
```

Out[ ]:
```
'Os aprovados foram André e José'
```

In [ ]:
```
# Nesse caso, embora sejam números, estão entre aspas e portanto, são lidos como strings
lista = ['28','29','30']
```

In [ ]:
```
lista[0]
```

Out[ ]:
```
'28'
```

In [ ]:
```
type(lista[0])
```

Out[ ]:
```
str
```

In [ ]:
```
'A minha idade é ' + lista[0]
```

Out[ ]:
```
'A minha idade é 28'
```

In [ ]:
```
lista[2]
```

Out[ ]:
```
'30'
```

In [ ]:
```
# As listas são mutáveis, ao contrário das strings

lista[2] = '31'
```

In [ ]:
```
lista
```

Out[ ]:
```
['28', '29', '31']
```

In [ ]:
```
# Também são aditivas

nova_lista = nomes + lista
```

In [ ]:
```
nova_lista
```

Out[ ]:
```
['André', 'João', 'José', '28', '29', '31']
```

In [ ]:
```
# Repetir a lista 2 vezes

nova_lista*2
```

Out[ ]:
```
['André',
 'João',
 'José',
 '28',
 '29',
 '31',
 'André',
 'João',
 'José',
 '28',
 '29',
 '31']
```

In [ ]:
```
# Adicionar um novo elemento à lista. Método 'append'

# Note que estou incluindo um item numérico (integer)

lista.append(30)
```

In [ ]:
```
lista
```

Out[ ]:
```
['28', '29', '31', 30]
```

In [ ]:
```
type(lista[3])
```

Out[ ]:
```
int
```

In [ ]:
```
# Método 'pop' remove o último item

lista.pop()
```

Out[ ]:
```
30
```

In [ ]:
```
lista
```

Out[ ]:
```
['28', '29', '31']
```

In [ ]:
```
lista.pop()
```

Out[ ]:
```
'31'
```

In [ ]:
```
lista
```

Out[ ]:
```
['28', '29']
```

In [ ]:
```
#Lista apenas com itens númericos

idade = [4, 54, 65, 34, 29, 78, 38]
```

In [ ]:
```
# Método sort para classificar por ordem crescente (no caso de strings será por ordem alfabética)

idade.sort()
```

In [ ]:
```
idade
```

Out[ ]:
```
[4, 29, 34, 38, 54, 65, 78]
```

In [ ]:
```
# Método reverse para classificar por ordem decrescente (ordem alfabética invertida)

idade.reverse()
```

In [ ]:
```
idade
```

Out[ ]:
```
[78, 65, 54, 38, 34, 29, 4]
```

In [ ]:
```
# Criar três listas

lista1 = [0,1,2]
lista2 = [4,5,6]
lista3 = [7,8,9]
```

In [ ]:
```
# Unir as três listas em uma lista só (lista de listas)

matriz = [lista1, lista2, lista3]
```

In [ ]:
```
matriz
```

Out[ ]:
```
[[0, 1, 2], [4, 5, 6], [7, 8, 9]]
```

In [ ]:
```
# Acessar a lista indexada como 1 (segunda lista) e seu item 0 (primeiro item)

matriz[1][0]
```

Out[ ]:
```
4
```

In [ ]:
```
# Primeira lista e primeiro elemento

matriz[0][0]
```

Out[ ]:
```
0
```

6. Dicionários

Os dicionários representam uma estrutura de dados mais complexos que as listas, capazes de relacionar pares de dados (um dado que pode representar outro dado). Ao invés de colchetes, como as listas, são definidos pelas chaves. Em outras linguagens, poderia ser relacionado às "arrays associativas". A estrutura básica dos dicionários é "{*key: value*}"

In [ ]:
```
precos = {'PETR4':30,
          'VALE3': 102,
          'WEGE3': 34}
```

In [ ]:
```
precos
```

Out[ ]:
```
{'PETR4': 30, 'VALE3': 102, 'WEGE3': 34}
```

In [ ]:
```
type(precos)
```

Out[ ]:
```
dict
```

In [ ]:
```
# Listar apenas as chaves (keys) do dicionário

precos.keys()
```

Out[ ]:
```
dict_keys(['PETR4', 'VALE3', 'WEGE3'])
```

In [ ]:
```
# Listar apenas os valores do dicionário

precos.values()
```

Out[ ]:
```
dict_values([30, 102, 34])
```

In [ ]:
```
# Listar os pares de itens do dicionário

precos.items()
```

Out[ ]:
```
dict_items([('PETR4', 30), ('VALE3', 102), ('WEGE3', 34)])
```

In [ ]:
```
# Indexação é feito pelas chaves e não pelos índices numéricos (como listas)

precos['PETR4']
```

Out[ ]:
```
30
```

In [ ]:
```
precos['VALE3']
```

Out[ ]:
```
102
```

In [ ]:
```
# Os dicionários são bem flexíveis quanto aos "values". Poderia ser uma string, uma lista...

precos = {'PETR4':'30',
          'VALE3': [102,103,104,105],
          'WEGE3': 34}
```

In [ ]:
```
precos['PETR4']
```

Out[ ]:
```
'30'
```

In [ ]:
```
precos['VALE3']
```

Out[ ]:
```
[102, 103, 104, 105]
```

In [ ]:
```
# São mutáveis, assim como as listas

precos['WEGE3'] = 30
```

In [ ]:
```
precos
```

Out[ ]:
```
{'PETR4': '30', 'VALE3': [102, 103, 104, 105], 'WEGE3': 30}
```

In [ ]:
```
# Também permitem adição de novos itens

precos['GOAU4'] = 20
```

In [ ]:
```
precos
```

Out[ ]:
```
{'PETR4': '30', 'VALE3': [102, 103, 104, 105], 'WEGE3': 30, 'GOAU4': 20}
```

In [ ]:
```
listas = []
```

In [ ]:
```
dicionarios = {}
```

7. Tuplas

As tuplas são criadas utilizando parênteses. Indexação e slicing similares aos de strings e listas. Estão entre as estruturas de dados mais simples (ex., são imutáveis e nao permitem adições), porém com pouca flexibilidade.

In [ ]:
```
tupla = ()
```

In [ ]:
```
# Exemplo de uma tupla (permite valores repetidos)

tupla1 = (1,1,2,2,4,6,15)
```

In [ ]:
```
tupla1
```

Out[ ]:
```
(1, 1, 2, 2, 4, 6, 15)
```

In [ ]:
```
type(tupla1)
```

Out[ ]:
```
tuple
```

In [ ]:
```
# Comprimento da tupla = número de itens

len(tupla1)
```

Out[ ]:
```
7
```

In [ ]:
```
# Indexação similar a listas

tupla1[2]
```

Out[ ]:
```
2
```

In [ ]:
```
# Fatiamento similar às listas e strings

tupla1[2:6]
```

Out[ ]:
```
(2, 2, 4, 6)
```

In [ ]:
```
# Tuplas possuem alguns métodos simples, como determinar o índice referente a um item específico (itens repet

tupla1.index(15)
```

Out[ ]:
```
6
```

In [ ]:
```
tupla1.index(1)
```

Out[ ]:
```
0
```

In [ ]:
```
# Método contagem de itens iguais

tupla1.count(15)
```

Out[ ]:
```
1
```

In [ ]:
```
# Elas são imutáveis

# Este comando vai resultar em erro

tupla1[0] = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-67-b04eb0cc8ee2> in <module>
      3 # Este comando vai resultar em erro
      4
----> 5 tupla1[0] = 3

TypeError: 'tuple' object does not support item assignment
```

In [ ]:
```
tupla1[0]
```

Out[ ]:
```
1
```

8. Conjuntos

Um conjunto (*set*), assim como a tupla, é criado utilizando parênteses. Não existe indexação ou slicing, ou seja, não seguem uma ordem (nesse caso mais similares aos dicionários). Permitem adições mas não permitem itens repetidos.

In [ ]:
```
# Cria-se um conjunto a partir da função set()

conjunto1 = set()
```

In [ ]:
```
conjunto1
```

Out[ ]:
```
set()
```

In [ ]:
```
type(conjunto1)
```

Out[ ]:
```
set
```

In [ ]:
```
# Método add para adicionar um novo item

conjunto1.add(9)
```

In [ ]:
```
conjunto1
```

Out[ ]:
```
{9}
```

In [ ]:
```
# Não consideram itens repetidos (pode ser usado para remover duplicatas)

# Veja que vamos adicionar o mesmo número várias vezes, mas ele os contabiliza apenas uma vez

conjunto1.add(9)
conjunto1.add(9)
conjunto1.add(9)
conjunto1.add(1)
conjunto1.add(1)
conjunto1.add(2)
conjunto1.add("PETR4")
```

In [ ]:
```
conjunto1
```

Out[ ]:
```
{1, 2, 9, 'PETR4'}
```

In [ ]:
```
len(conjunto1)
```

Out[ ]:
```
4
```

In [ ]:
```
# Não são indexáveis, ou seja, não seguem uma ordem como as listas ou tuplas

# Portanto, este comando vai resultar em erro

conjunto1[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-79-e89d6ee110e5> in <module>
      3 # Portanto, este comando vai resultar em erro
      4
----> 5 conjunto1[0]

TypeError: 'set' object is not subscriptable
```

9. Declarações condicionais

Em linguagens de programação, frequentemente devemos criar estruturas e comandos para atuar de acordo com condições específicas, ou seja, apenas se alguma condição for satisfeita.

In [ ]:
```
# Exemplo: Determinar o preço atual de um ativo e o seu preço alvo de compra

preco_azul = 35.28
preco_alvo = 30
```

In [ ]:
```
# Comando if ("se", em portugues), estabelecendo uma declaração condicional. Importante destacar a indentação
# elif é uma contração do else if

if preco_azul < 30:
    print('Compre, preço muito descontado!')
elif preco_azul < 33:
    print('Ainda em tendência de queda, espere um pouco mais!')
elif preco_azul <= 36:
    print('Iniciou tendência de queda, espere cair até 30')
else:
    print('Espere!')
```

```
Iniciou tendência de queda, espere cair até 30
```

In [ ]:
```
preco_weg = 19
```

In [ ]:
```
preco_alvo = 33
```

In [ ]:
```
# Ressaltando a estrutura da declaração condicional -> if seguido da condição e ":" no fim.
# Na linha seguinte, identação e comando caso a condição seja verdadeira

if preco_weg > preco_alvo:
    print('Espere')
else:
    print('Compre!')
```

```
Compre!
```

In [ ]:
```
# Importante observar que as condições vão sendo checadas pela ordem em que foram declaradas.
# Assim que uma delas é satisfeita, as demais não são executadas mais

if preco_weg < 25:
    print('Compre, preço muito descontado!')
elif preco_weg < 30:
    print('Ainda em tendência de queda, espere um pouco mais!')
elif preco_weg <= 32:
    print('Iniciou tendência de queda, espere cair até 25')
else:
    print('Espere!')
```

```
Compre, preço muito descontado!
```

In [ ]:
```
# Trocando a ordem das declarações. Note que assim que atendeu a primeira, as demais são ignoradas

if preco_weg < 30:
```

```
    print('Ainda em tendência de queda, espere um pouco mais!')
elif preco_weg < 25:
    print('Compre, preço muito descontado!')
elif preco_weg <= 32:
    print('Iniciou tendência de queda, espere cair até 25')
else:
    print('Espere!')
```

```
Ainda em tendência de queda, espere um pouco mais!
```

10. Estruturas de repetição

Além das estruturas (ou declarações) condicionais, as estruturas de repetição são muito importantes na sintaxe de qualquer linguagem de programação. Imagine que você queira automatizar uma tarefa, informando que ela deve ocorrer por determinadas vezes ou até que outra alguma outra condição seja estabelecida

10.1. For

In [ ]:
```
lista = ['João', 'José', 'Andre', 'Marcos']
```

In [ ]:
```
# Para cada item dentro da lista, imprima o item. Essa automação também é conhecida como loop

for item in lista:
    print(item)
```

```
João
José
Andre
Marcos
```

In [ ]:
```
# Embora tenha usado a palavra item, o Python compreende que estou determinando elementos/itens de uma lista.

for proprietario_automovel in lista:
    print(proprietario_automovel)
```

```
João
José
Andre
Marcos
```

In [ ]:
```
lista_ativos = ['PETR4.SA', 'WEGE3.SA', 'MGLU3.SA', 'LREN3.SA', 'RENT3.SA', 'VALE3.SA']
```

In [ ]:
```
lista_ativos
```

Out[ ]:
```
['PETR4.SA', 'WEGE3.SA', 'MGLU3.SA', 'LREN3.SA', 'RENT3.SA', 'VALE3.SA']
```

In [ ]:
```
# Criando uma automação para que pegue cada item da lista e adicione a frase "é um ativo da B3".
for acoes in lista_ativos:
    print(acoes,"é um ativo da B3")
```

```
PETR4.SA é um ativo da B3
WEGE3.SA é um ativo da B3
MGLU3.SA é um ativo da B3
LREN3.SA é um ativo da B3
RENT3.SA é um ativo da B3
VALE3.SA é um ativo da B3
```

In [ ]:
```
# Automação usando a função break para interromper o loop assim que uma condição for satisfeita (integrando '
# Atenção à identação correta

for acoes in lista_ativos:
    print(acoes,"é um ativo da B3")
    if acoes == "LREN3.SA":
        break
```

```
PETR4.SA é um ativo da B3
WEGE3.SA é um ativo da B3
MGLU3.SA é um ativo da B3
LREN3.SA é um ativo da B3
```

In [ ]:
```
for acoes in lista_ativos:
    print(acoes,"é um ativo da B3")
    if acoes == "WEGE3.SA":
        break
```

```
PETR4.SA é um ativo da B3
WEGE3.SA é um ativo da B3
```

In [ ]:
```
precos = {'PETR4':30,
          'VALE3': 102,
```

```
          'WEGE3': 34}
```

In [ ]:
```
len(precos)
```

Out[ ]:
```
3
```

In [ ]:
```
for i in range(0,len(precos)):
    print(list(precos.keys())[i],"está com o preço de",list(precos.values())[i],"neste momento")
```

```
PETR4 está com o preço de 30 neste momento
VALE3 está com o preço de 102 neste momento
WEGE3 está com o preço de 34 neste momento
```

10.2. While

A estrutura do while (enquanto, em português) se refere a outra repetição (loop) que ocorrerá enquanto a condição for verdadeira. Poderia ser pensada de uma maneira análoga ao uso do 'for' e 'if' integrados.

In [ ]:
```
preco_weg = 30
```

In [ ]:
```
# Enquanto o preço for menor que 39, imprima a frase "Ainda não atingiu o preço alvo, espere. O preço atual é
# Caso a condição pare de ser verdadeira (preço_weg seja >= 39) imprima esta outra frase: "Preço alvo atingid

while preco_weg < 39:
    print("Ainda não atingiu o preço alvo, espere. O preço atual é: ", round(preco_weg,2))
    preco_weg = preco_weg*1.01

else:
    print("Preço alvo atingido, hora de vender! Preço da venda: ", round(preco_weg, 2))
```

```
Ainda não atingiu o preço alvo, espere. O preço atual é:  30
Ainda não atingiu o preço alvo, espere. O preço atual é:  30.3
Ainda não atingiu o preço alvo, espere. O preço atual é:  30.6
Ainda não atingiu o preço alvo, espere. O preço atual é:  30.91
Ainda não atingiu o preço alvo, espere. O preço atual é:  31.22
Ainda não atingiu o preço alvo, espere. O preço atual é:  31.53
Ainda não atingiu o preço alvo, espere. O preço atual é:  31.85
Ainda não atingiu o preço alvo, espere. O preço atual é:  32.16
Ainda não atingiu o preço alvo, espere. O preço atual é:  32.49
Ainda não atingiu o preço alvo, espere. O preço atual é:  32.81
Ainda não atingiu o preço alvo, espere. O preço atual é:  33.14
Ainda não atingiu o preço alvo, espere. O preço atual é:  33.47
Ainda não atingiu o preço alvo, espere. O preço atual é:  33.8
Ainda não atingiu o preço alvo, espere. O preço atual é:  34.14
Ainda não atingiu o preço alvo, espere. O preço atual é:  34.48
Ainda não atingiu o preço alvo, espere. O preço atual é:  34.83
Ainda não atingiu o preço alvo, espere. O preço atual é:  35.18
Ainda não atingiu o preço alvo, espere. O preço atual é:  35.53
Ainda não atingiu o preço alvo, espere. O preço atual é:  35.88
Ainda não atingiu o preço alvo, espere. O preço atual é:  36.24
Ainda não atingiu o preço alvo, espere. O preço atual é:  36.61
Ainda não atingiu o preço alvo, espere. O preço atual é:  36.97
Ainda não atingiu o preço alvo, espere. O preço atual é:  37.34
Ainda não atingiu o preço alvo, espere. O preço atual é:  37.71
Ainda não atingiu o preço alvo, espere. O preço atual é:  38.09
Ainda não atingiu o preço alvo, espere. O preço atual é:  38.47
Ainda não atingiu o preço alvo, espere. O preço atual é:  38.86
Preço alvo atingido, hora de vender! Preço da venda:  39.25
```

In [ ]:
```
# Outro exemplo de uso do while. Imagine que tem disponível 10000 reais para investir. A cada operação, sera
# A função input interage com o usuario, recebendo um objeto (numérico, string...) e armazenando em uma variá

investimento_unitario = 0
investimento_total = 0
```

In [ ]:
```
# Enquanto o valor alocado for menor que 10000 a condição do while será executada. Quando essa condição não f
while investimento_total < 10000:
    investimento_unitario = int(input("Digite o valor investido na operação: "))
    investimento_total = investimento_total + investimento_unitario

else:
    print("Limite total previsto para investir foi atingido!")
    print("Você excedeu ", investimento_total-10000, " do valor previsto")

print("Você investiu o total de R$", investimento_total)
```

```
Digite o valor investido na operação: 10000
Limite total previsto para investir foi atingido!
Você excedeu  0  do valor previsto
Você investiu o total de R$ 10000
```

11. Alguns outros operadores

Alguns operadores são muito úteis para execução de estruturas condicionais e de repetição. Dentre eles:

11.1. Range

Indica um intervalo entre dois números. Segue o padrão da indexação (primeiro argumento inclusivo e segundo argumento não inclusivo)

In [ ]:
```
# Intervalo entre 0 (inclusivo) e 11 (exclusivo)
# O output não lista todos os números, reduzindo gastos com memória

range(0,11)
```

Out[ ]:
```
range(0, 11)
```

In [ ]:
```
# Nessa caso, listará todos os elementos

list(range(0,11))
```

Out[ ]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [ ]:
```
# Compreende todo o intervalo sequencial

for carros in range(0,11):
    print(carros)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

In [ ]:
```
# Também pode ser usado especificando um intervalo entre os números dentro do intervalo estabelecido, similar

for i in range(0,11,2):
    print(i)
```

```
0
2
4
6
8
10
```

11.2. Enumerate

Função que transforma uma coleção de dados (lista, tupla, string) em um objeto enumerado de cada item da coleção

In [ ]:
```
lista_ativos = ['PETR4.SA', 'WEGE3.SA', 'MGLU3.SA', 'LREN3.SA', 'RENT3.SA', 'VALE3.SA']
```

In [ ]:
```
# O output não enumera todos os itens, reduzindo gastos com memória

enumerate(lista_ativos)
```

Out[ ]:
```
<enumerate at 0x24cc55342c0>
```

In [ ]:
```
# Nessa caso, irá enumerar todos os elementos

list(enumerate(lista_ativos))
```

Out[ ]:
```
[(0, 'PETR4.SA'),
 (1, 'WEGE3.SA'),
 (2, 'MGLU3.SA'),
 (3, 'LREN3.SA'),
 (4, 'RENT3.SA'),
 (5, 'VALE3.SA')]
```

In [ ]:
```
# Podemos começar a enumeração a partir de outro número que não o 0

list(enumerate(lista_ativos,3))
```

Out[ ]:
```
[(3, 'PETR4.SA'),
```

```
 (4, 'WEGE3.SA'),
 (5, 'MGLU3.SA'),
 (6, 'LREN3.SA'),
 (7, 'RENT3.SA'),
 (8, 'VALE3.SA')]
```

In [ ]:
```
# Exemplo usando uma string

acao = 'PETR4'
```

In [ ]:
```
list(enumerate(acao))
```

Out[ ]:
```
[(0, 'P'), (1, 'E'), (2, 'T'), (3, 'R'), (4, '4')]
```

In [ ]:
```
for i, letra in enumerate(acao):
    print("No índice", i, "o caractere é", letra)
```

```
No índice 0 o caractere é P
No índice 1 o caractere é E
No índice 2 o caractere é T
No índice 3 o caractere é R
No índice 4 o caractere é 4
```

In [ ]:
```
# Note que neste caso, o enumerate fez a mesma função da estrutura 'for' a seguir
index_count = 0

for letra in acao:
    print("No índice", index_count, "o caractere é", letra)
    index_count += 1
```

```
No índice 0 o caractere é P
No índice 1 o caractere é E
No índice 2 o caractere é T
No índice 3 o caractere é R
No índice 4 o caractere é 4
```

11.3. Zip & in

A função *zip* consegue executar interações entre listas. O operador *in* (dentro/pertencente em português) permite a leitura sequencial de objetos zip (assim como de listas), fundamental nas estruturas 'for'

In [ ]:
```
acoes = ['PETR4', 'WEGE3', 'MGLU3', 'JHSF3', 'GOAU4']
setores = ['Petróleo', 'Motores', 'Varejo', 'Construção', 'Metalurgia']
```

In [ ]:
```
# Criar uma relação entre as duas listas acima. Output reduzido assim como os anteriores range e enumerate

zip(acoes, setores)
```

Out[ ]:
```
<zip at 0x24cc543e840>
```

In [ ]:
```
# Note que é uma lista de tuplas

list(zip(acoes, setores))
```

Out[ ]:
```
[('PETR4', 'Petróleo'),
 ('WEGE3', 'Motores'),
 ('MGLU3', 'Varejo'),
 ('JHSF3', 'Construção'),
 ('GOAU4', 'Metalurgia')]
```

In [ ]:
```
# Leitura sequencial do zip usando o in

for acao, setor in list(zip(acoes, setores)):
    print('O papel', acao, 'pertence ao setor', setor)
```

```
O papel PETR4 pertence ao setor Petróleo
O papel WEGE3 pertence ao setor Motores
O papel MGLU3 pertence ao setor Varejo
O papel JHSF3 pertence ao setor Construção
O papel GOAU4 pertence ao setor Metalurgia
```

In [ ]:
```
# Mostrando o operador f'', que será útil no exemplo seguinte
# Esse operador admite que usemos variáveis dentro das aspas, desde que essas variáveis estejam por sua vez d
# de chaves, como no exemplo abaixo

papel = 'PETR3'
setor = 'Petróleo'
```

In [ ]:
```
f'A {papel} está no setor de {setor}'
```

Out[ ]: 'A PETR3 está no setor de Petróleo'

In [ ]:
```
# Uma outra forma de escrever o mesmo loop acima, desta dez usando o operador f''

for acao, setor in list(zip(acoes, setores)):
    print(f'O papel {acao} pertence ao setor {setor} ')
```

```
O papel PETR4 pertence ao setor Petróleo
O papel WEGE3 pertence ao setor Motores
O papel MGLU3 pertence ao setor Varejo
O papel JHSF3 pertence ao setor Construção
O papel GOAU4 pertence ao setor Metalurgia
```

In [ ]:
```
# Além de leitura sequencial, o in pode ser usado como operador lógico

'VIVR3' in acoes
```

Out[ ]: False

In [ ]:
```
'WEGE3' in acoes
```

Out[ ]: True

11.4. Random

Este operador pode ser utilizado para obter números aleatórios de acordo com alguma condição pré-determinada

In [ ]:
```
from random import randint, shuffle
```

A função randint() retorna um número aleatório dentro de um intervalo específico

In [ ]:
```
randint(0,100)
```

Out[ ]: 67

In [ ]:
```
# Vamos gerar uma lista com 10 números aleatórios que sejam entre 10 e 100

lista_rand = []

for i in range(0,10):
    lista_rand.append(randint(0,100))
```

In [ ]:
```
lista_rand
```

Out[ ]: [85, 97, 24, 47, 78, 13, 91, 95, 21, 63]

In [ ]:
```
# Utilizando o operador shuffle para 'embaralhar' os números
shuffle(lista_rand)
```

In [ ]:
```
lista_rand
```

Out[ ]: [24, 78, 21, 91, 95, 47, 63, 13, 97, 85]

11.5. min e max

Funções para retornar os valores mínimo e máximo de uma lista

In [ ]:
```
min(lista_rand)
```

Out[ ]: 13

In [ ]:
```
max(lista_rand)
```

Out[ ]: 97

11.6. input

Função que permite que o usuário inpute algum valor de acordo com o que é pedido por um texto e armazene isso na memória de uma variável

In [ ]:
```
acao = input('Por favor insira a ação que deseja comprar: ')
```

```
Por favor insira a ação que deseja comprar: PETR4
```

In [ ]:
```
acao
```

Out[ ]: 'PETR4'

In [ ]:
```
print(f'O cliente deseja comprar o papel {acao} ')
```

```
O cliente deseja comprar o papel PETR4
```

12. List Comprehension (Compreensão de Lista)

Forma de executar loops com uma estrutura de código mais resumida do que os "for" tradicionais

In [ ]:
```
acao = 'MGLU3'
```

Imagine que precisamos realizar a tarefa de transformar cada caractere na string acima em um item separado dentro de uma nova lista.

Como feríamos isso com nosso conhecimento em Python até agora?

Exemplo 01

Na forma tradicional usando "for", faríamos assim:

In [ ]:
```
lista_caracteres = []
```

In [ ]:
```
for caractere in acao:
    lista_caracteres.append(caractere)
```

In [ ]:
```
lista_caracteres
```

Out[ ]: ['M', 'G', 'L', 'U', '3']

In [ ]:
```
# Podemos usar o indexing para retornar algum caractere específico

lista_caracteres[1]
```

Out[ ]: 'G'

E como fazer isso utilizando a compreensão de lista?

In [ ]:
```
# Perceba que o comando ficou bem mais simples

[caractere for caractere in acao]
```

Out[ ]: ['M', 'G', 'L', 'U', '3']

O output da compreensão de lista é uma nova lista

Exemplo 02

Imagine que você recebeu uma série de tickers que vieram incompletos. Antes de realizar sua rotina, precisa adicionar um número '3' aos nomes dos papéis. Como você faria?

In [ ]:
```
ativos = ['MGLU', 'VALE', 'WEGE', 'LREN']
```

In [ ]:
```
# Criamos uma lista vazia que será populada com os novos nomes dos ativos
novos_ativos = []

# Para cada ativo, vamos adicionar o caractere '3' no final
for letra in ativos:
    novos_ativos.append(letra + '3')
```

In [ ]:
```
novos_ativos
```

Out[ ]: ['MGLU3', 'VALE3', 'WEGE3', 'LREN3']

Podemos realizar essa tarefa com uma compreensão de lista de uma forma mais rápida.

In [ ]:
```
[letra + '3' for letra in ativos]
```

Out[ ]: ['MGLU3', 'VALE3', 'WEGE3', 'LREN3']

Exemplo 03

Imagine que quiséssemos elevar todos os números de 0 a 10 ao quadrado.

Da forma tradicional, faríamos:

In [ ]:
```
list_num = []

for num in range(0,11):
    list_num.append(num**2)
```

In [ ]:
```
# Resultado

list_num
```

Out[ ]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Com a compreensão de lista, podemos fazer

In [ ]:
```
[num**2 for num in range(0,11)]
```

Out[ ]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

13. Funções e métodos

Uma função é um comando capaz de realizar uma tarefa, de acordo com critérios que determinamos.

A função só é executada de fato quando é chamada, e executa atividades que foram especificadas pelo usuário.

Por exemplo: abaixo estamos criando uma função chamada "tempo_hoje", que é executada quando chamada. Seu propósito é imprimir a mensagem "Hoje faz sol" quando chamada.

In [ ]:
```
def tempo_hoje():
    print("Hoje faz sol")
```

In [ ]:
```
tempo_hoje()
```

```
Hoje faz sol
```

Vamos criar agora uma nova função que vai retornar a soma de dois números, sendo estes determinados pelo usuário.

In [ ]:
```
# Perceba que a função abaixo recebe dois arguentos: num1 e num2. A função vai então somar num1 e num2 e reto

def soma_numeros(num1, num2):
    soma = num1 + num2
    return soma
```

Vamos passar nessa função os números 2 e 3

In [ ]:
```
soma_numeros(2, 3)
```

Out[ ]:
```
5
```

In [ ]:
```
novo_objeto = soma_numeros(1000, 3050)
```

In [ ]:
```
novo_objeto
```

Out[ ]:
```
4050
```

Vamos criar agora uma nova função chamada "preco" que basicamente retorna a cotação de um papel, de acordo com o que for especificado pelo usuário.

In [ ]:
```
def preco(acao):
    if (acao == 'PETR4'):
        print('O preço agora é R$ 30,89')
    elif (acao == 'VALE3'):
        print('O preço agora é R$ 112')
    elif (acao == 'WEGE3'):
        print('O preço agora é R$ 32')
    elif (acao == 'MGLU3'):
        print('O preço agora é R$ 6,79')
    else:
        print('Papel desconhecido')
```

In [ ]:
```
preco('PETR4')
```

```
O preço agora é R$ 30,89
```

In [ ]:
```
preco('MGLU3')
```

```
O preço agora é R$ 6,79
```

Criando Docstring da função

O que é?

"Docstring" é a documentação da função, ou seja, a mensagem que aparece quando o usuário consulta a ajuda da função.

In [ ]:
```python
def preco(acao):

    ''' ESSA FUNÇÃO DEVE SER UTILIZADA PARA RETORNAR A COTAÇÃO EM TEMPO REAL DE UM PAPEL '''

    if (acao == 'PETR4'):
        print('O preço agora é R$ 30,89')
    elif (acao == 'VALE3'):
        print('O preço agora é R$ 112')
    elif (acao == 'WEGE3'):
        print('O preço agora é R$ 32')
    elif (acao == 'MGLU3'):
        print('O preço agora é R$ 24,59')
    else:
        print('Papel desconhecido')
```

Métodos

Métodos são funções aplicáveis apenas a determinados tipos de objetos. Todos os métodos são funções, mas nem todas as funções são métodos. Os métodos são executados ao fim dos objetos e separados com um ponto.

In [ ]:
```python
numeros = [1, 2, 4, 6, 10]
```

Perceba que a função append() abaixo é um método.

In [ ]:
```python
numeros.append(5)
```

In [ ]:
```python
numeros
```

Out[ ]:
```
[1, 2, 4, 6, 10, 5]
```

14. Funções embutidas

Também conhecidas como "built-in functions" são funções nativas do Python que não necessitam de nenhuma biblioteca para sua execução.

14.1. Map

A funçã map() aplica um comando (função) a vários elementos de uma lista, de uma vez.

Confira alguns exemplos:

In [ ]:
```python
def potencia(num, pot = 2):
    return num**pot
```

In [ ]:
```python
potencia(3)
```

Out[ ]:
```
9
```

In [ ]:
```python
potencia(3, 4)
```

Out[ ]:
```
81
```

In [ ]:
```python
lista_numeros = [1, 2, 3, 4, 5]
```

In [ ]:
```python
list(map(potencia, lista_numeros))
```

Out[ ]:
```
[1, 4, 9, 16, 25]
```

14.2. Filter

De forma análoga, a função "filter" aplica um filtro a vários elementos de uma lista de uma só vez.

In [ ]:
```python
# Vamos criar uma função que retorna uma condição lógica, dizendo se um número é par (True) ou não (False)

def checar_par(num):
    return num%2 ==0
```

In [ ]:
```
checar_par(2)
```

Out[ ]:
```
True
```

In [ ]:
```
checar_par(9)
```

Out[ ]:
```
False
```

Se aplicarmos essa função à lista criada no item anterior, ela vai trazer apenas os números que são pares.

In [ ]:
```
list(filter(checar_par, lista_numeros))
```

Out[ ]:
```
[2, 4]
```

15. Funções anônimas: lambda

São funções que não precisam de definição forma (explícita) podendo ser chamadas em apenas uma linha.

Suponha que queremos criar uma função que eleva os números ao quadrado:

In [ ]:
```
def quadrado(num):
    return num**2
```

In [ ]:
```
quadrado(5)
```

Out[ ]:
```
25
```

Com o comando lambda, não precisamos dar um nome a essa função nem defini-la, basta especificar qual a tarefa que precisamos executar

In [ ]:
```
lambda num: num**2
```

Out[ ]:
```
<function __main__.<lambda>>
```

Caso seja do nosso interesse, podemos criar uma variável que recebe a operação realizada pela função lambda.

In [ ]:
```
quadrado_lambda = lambda num: num**2
```

In [ ]:
```
quadrado_lambda(5)
```

Out[ ]:
```
25
```

Podemos ir ainda mais longe, aplicando essas funções a vários elementos de uma lista, com as funções **map** e **filter**

In [ ]:
```
list(map(lambda num: num**2, lista_numeros))
```

Out[ ]:
```
[1, 4, 9, 16, 25]
```

In [ ]:
```
list(filter(lambda num: num%2 ==0, lista_numeros))
```

Out[ ]:
```
[2, 4]
```

16. Escopo das variáveis

Podemos entender o escopo como a "área de atuação" de uma variável.

In [ ]:
```
# escopo global

x = 25
```

In [ ]:
```
def printer():
    x = 50
    return x
```

In [ ]:
```
x
```

Out[ ]:
```
25
```

In [ ]:
```
printer()
```

Out[ ]:
```
50
```

17. Args

Imagine que criamos uma função nova e especificamos a quantidade de argumentos que precisam ser passados nessa função:

In [ ]:
```
# Neste caso estamos criando uma função que recebe três argumentos: a, b e c

def soma_varios(a,b,c):
    soma = sum((a,b,c))
    return soma
```

In [ ]:
```
soma_varios(1,2,3)
```

Out[ ]:
```
6
```

In [ ]:
```
# Com 5 argumentos

def soma_varios(a,b,c,d,e):
    soma = sum((a,b,c,d,e))
    return soma
```

In [ ]:
```
soma_varios(1,2,3,4,5)
```

Out[ ]:
```
15
```

In [ ]:
```
# E num caso de ter N argumentos (número indefinido), como seria?
# Graças ao argumento *args, podemos dar a opção da função receber inúmeros argumentos

def soma_varios(*args):
    soma = sum(args)
    return soma
```

In [ ]:
```
soma_varios(1)
```

Out[ ]:
```
1
```

In [ ]:
```
soma_varios(1,2,3)
```

Out[ ]:
```
6
```

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

Out[ ]:
```
45
```

18. Kwargs

De forma análoga ao *args, o **kwargs torna a função capaz de receber argumentos que tem um nome

In [ ]:
```
# No caso abaixo, o nome do argumento é 'acao'

def compra_de_acoes(**kwargs):
    if 'acao' in kwargs:
        print('A ação comprada foi ', kwargs['acao'])
    else:
        print('O cliente não investe em ações')
```

In [ ]:
```
compra_de_acoes(acao = 'PETR4')
```

```
A ação comprada foi  PETR4
```

In [ ]:
```
compra_de_acoes()
```

```
O cliente não investe em ações
```

In [ ]:
```
# No caso abaixo, temos dois argumentos do tipo kwargs que pedem um nome: 'acao' e 'lote'

def montagem_posicao(*args,**kwargs):
    if 'acao' and 'lote' in kwargs:
        print('O cliente montou uma posição de ', ' e '.join(kwargs['acao']))
        print('Ele comprou ', ' e '.join(args), 'papéis, respectivamente')
        print('Todas elas foram compradas no lote', kwargs['lote'])
```

In [ ]:
```
montagem_posicao('100', '200', acao = ('PETR4', 'VALE3'), lote = 'inteiro')
```

```
O cliente montou uma posição de  PETR4 e VALE3
```

```
Ele comprou  100 e 200 papéis, respectivamente
Todas elas foram compradas no lote inteiro
```

19. Trabalhando com bibliotecas

1. O que são bibliotecas
2. Importância de se utilizar bibliotecas

19.1. Importando as bibliotecas

Nessa seção, vamos estudar as duas bibliotecas mais importantes do Python: pandas e numpy

In [ ]:
```
import numpy as np
import pandas as pd
```

19.2. Numpy

O principal tipo de objeto que criamos com a numpy é o 'array'

Para criar arrays, temos diversas funções diferentes, cada um servido a um propósito específico

Por exemplo, podemos criar arrays que contém apenas zeros

In [ ]:
```
a = np.zeros(3)
```

In [ ]:
```
a
```

Out[ ]:
```
array([0., 0., 0.])
```

In [ ]:
```
# O atributo .shape retorna as dimensões do array

a.shape
```

Out[ ]:
```
(3,)
```

In [ ]:
```
# As dimensões são manipuláveis. Logo, podemos especificar o número de linhas e colunas:

a.shape = (3, 1)
```

In [ ]:
```
a.shape
```

Out[ ]:
```
(3, 1)
```

In [ ]:
```
a
```

Out[ ]:
```
array([[0.],
       [0.],
       [0.]])
```

Vamos criar um array contendo apenas '1'

In [ ]:
```
b = np.ones(10)
```

In [ ]:
```
b
```

Out[ ]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [ ]:
```
# Array vazio

c = np.empty(4)
```

In [ ]:
```
# veja que mesmo sendo próximo a 'vazio', os elementos da lista são não-nulos
c
```

Out[ ]:
```
array([6.23034601e-307, 9.34609790e-307, 1.33509389e-306, 6.67567993e-308])
```

In [ ]:
```
# Podemos criar um array determinando início, fim e número de componentes

d = np.linspace(3, 15, 10)
```

In [ ]:
```
d
```

Out[ ]:
```
array([ 3.        ,  4.33333333,  5.66666667,  7.        ,  8.33333333,
        9.66666667, 11.        , 12.33333333, 13.66666667, 15.        ])
```

In [ ]:
```
# Podemos especificar manualmente os elementos do array

e = np.array(10)
```

In [ ]:
```
e
```

Out[ ]:
```
array(10)
```

In [ ]:
```
f = np.array([1,2,3,4,5])
```

In [ ]:
```
f
```

Out[ ]:
```
array([1, 2, 3, 4, 5])
```

In [ ]:
```
# Uma lista pode virar um array!

lista = [4,5,6,7,8,9]
```

In [ ]:
```
g = np.array(lista)
```

In [ ]:
```
g
```

Out[ ]:
```
array([4, 5, 6, 7, 8, 9])
```

In [ ]:
```
type(g)
```

Out[ ]:
```
numpy.ndarray
```

In [ ]:
```
lista_listas = [[1, 2, 3, 4, 5], [4, 5, 6, 7, 8]]
```

In [ ]:
```
# Podemos usar mais de uma lista na construção do array:

h = np.array([lista_listas])
```

In [ ]:
```
h
```

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

In [ ]:
```
np.random.seed(0)
```

In [ ]:
```
i = np.random.randint(10, size = 6)
```

In [ ]:
```
i
```

Out[ ]:
```
array([5, 0, 3, 3, 7, 9])
```

Podemos realizar indexação e slicing nos arrays:

In [ ]:
```
i[0]
```

Out[ ]:
```
5
```

In [ ]:
```
i[3]
```

Out[ ]:
```
3
```

In [ ]:
```
i[-1]
```

Out[ ]:
```
9
```

Muito parecido com listas

Operações matemáticas

In [ ]:
```
a = np.array([10,20,100,200,500])
b = np.array([3,4,5,6,7])

np.add(a, b)
```

Out[ ]:
```
array([ 13,  24, 105, 206, 507])
```

In [ ]:
```
a + b
```

Out[ ]:
```
array([ 13,  24, 105, 206, 507])
```

In [ ]:
```
np.subtract(a, b)
```

Out[ ]:
```
array([  7,  16,  95, 194, 493])
```

In [ ]:
```
a - b
```

Out[ ]:
```
array([  7,  16,  95, 194, 493])
```

In [ ]:
```
np.multiply(a, b)
```

Out[ ]:
```
array([  30,   80,  500, 1200, 3500])
```

In [ ]:
```
a*b
```

Out[ ]:
```
array([  30,   80,  500, 1200, 3500])
```

In [ ]:
```
np.divide(a,b)
```

Out[ ]:
```
array([ 3.33333333,  5.        , 20.        , 33.33333333, 71.42857143])
```

In [ ]:
```
a/b
```

Out[ ]:
```
array([ 3.33333333,  5.        , 20.        , 33.33333333, 71.42857143])
```

19.3. Pandas

Mostrar aqui como baixar os dados de WEGE3 do Yahoo Finance

Temos um arquivo .csv que foi disponibilizado junto com o material do curso

In [ ]:
```
dados = pd.read_csv('WEGE3.SA.csv')
```

Com o atributo shape podemos verificar as dimensões

In [ ]:
```
dados.shape
```

Out[ ]:
```
(993, 7)
```

Com o atributo dtypes podemos verificar o tipo das variáveis

In [ ]:
```
dados.dtypes
```

Out[ ]:
```
Date          object
Open         float64
High         float64
Low          float64
Close        float64
Adj Close    float64
Volume       float64
dtype: object
```

In [ ]:
```
# Tipo do objeto dados

type(dados)
```

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

In [ ]:
```
# O método head() permite ver as 5 primeiras linhas

dados.head()
```

Out[ ]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2017-07-24	7.392307	7.473076	7.261538	7.426923	7.051007	2638220.0
1	2017-07-25	7.476923	7.476923	7.307692	7.369230	6.996235	2505880.0
2	2017-07-26	7.323076	7.442307	7.276923	7.326923	6.956069	2112240.0
3	2017-07-27	7.388461	7.419230	7.223076	7.269230	6.901296	1910480.0
4	2017-07-28	7.261538	7.296153	7.165384	7.200000	6.835570	2866760.0

In [ ]:
```
# O método tail() permite ver as 5 últimas linhas

dados.tail()
```

Out[ ]:

	Date	Open	High	Low	Close	Adj Close	Volume
988	2021-07-15	35.000000	35.590000	34.900002	34.919998	34.919998	4807000.0
989	2021-07-16	35.040001	35.259998	34.209999	34.290001	34.290001	5380000.0
990	2021-07-19	34.000000	34.369999	33.599998	34.180000	34.180000	4756600.0
991	2021-07-20	34.049999	34.830002	33.869999	34.549999	34.549999	4042300.0
992	2021-07-21	34.750000	34.970001	34.099998	34.450001	34.450001	4315700.0

In [ ]:
```
# Usamos o comando .loc para filtrar pelo index

dados.loc[0]
```

Out[ ]:
```
Date          2017-07-24
Open             7.39231
High             7.47308
Low              7.26154
Close            7.42692
Adj Close        7.05101
Volume       2.63822e+06
Name: 0, dtype: object
```

In [ ]:
```
dados.iloc[0]
```

Out[ ]:
```
Date          2017-07-24
Open             7.39231
High             7.47308
Low              7.26154
Close            7.42692
Adj Close        7.05101
Volume       2.63822e+06
Name: 0, dtype: object
```

In [ ]:
```
# Além de filtrar uma linha, podemos filtrar células

dados.iloc[0,0]
```

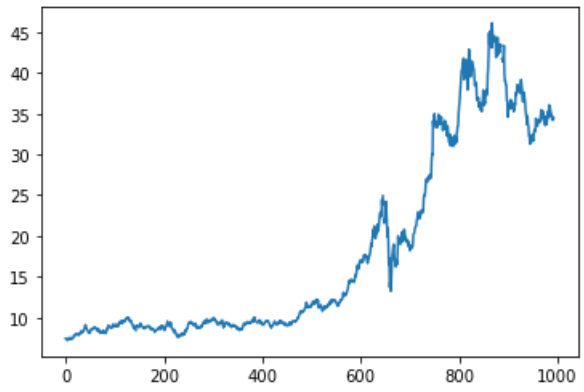
Out[ ]:
```
'2017-07-24'
```

In [ ]:
```
# Ou podemos filtrar colunas de nome específico

dados['Open']
```

Out[ ]:
```
0       7.392307
1       7.476923
2       7.323076
3       7.388461
4       7.261538
          ...
988    35.000000
989    35.040001
990    34.000000
991    34.049999
992    34.750000
Name: Open, Length: 993, dtype: float64
```

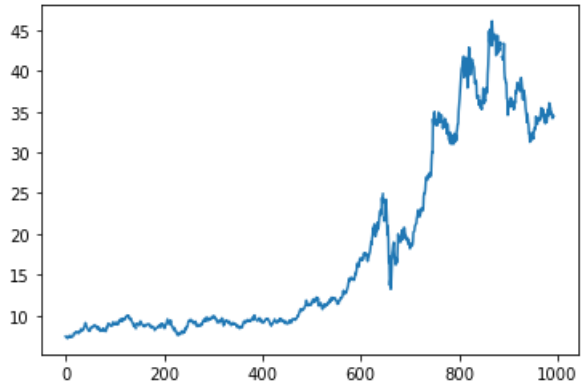
In [ ]:
```
# Podemos plotar apenas uma coluna

dados['Close'].plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x24cc8258520>
```



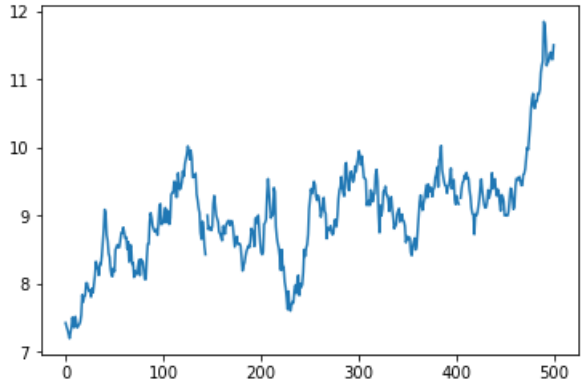
In [ ]:
```
dados.Close.plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x24cc8564820>
```



In [ ]:
```
# Podemos plotar um segmento de uma coluna. No exemplo abaixo, estamos plotando da linha 0 até a 500.

dados.iloc[0:500].Close.plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x24cc8611250>
```



In [ ]:
```
# Abaixo, estamos passando os valores da coluna 'Date' para o índice,
# de tal forma que o índice do dataframe se torne a própria data

dados.index = dados.Date
```

In [ ]:
```
dados.head()
```

Out[ ]:

Date	Date	Open	High	Low	Close	Adj Close	Volume
2017-07-24	2017-07-24	7.392307	7.473076	7.261538	7.426923	7.051007	2638220.0
2017-07-25	2017-07-25	7.476923	7.476923	7.307692	7.369230	6.996235	2505880.0
2017-07-26	2017-07-26	7.323076	7.442307	7.276923	7.326923	6.956069	2112240.0
2017-07-27	2017-07-27	7.388461	7.419230	7.223076	7.269230	6.901296	1910480.0
2017-07-28	2017-07-28	7.261538	7.296153	7.165384	7.200000	6.835570	2866760.0

In [ ]:
```
# Isso nos permite agora filtrar linhas de acordo com uma data específica

dados.loc['2017-07-24']
```

Out[ ]:
```
Date          2017-07-24
Open             7.39231
High             7.47308
Low              7.26154
Close            7.42692
Adj Close        7.05101
Volume       2.63822e+06
Name: 2017-07-24, dtype: object
```

Vamos trabalhar agora com diretórios diferentes

In [ ]:
```
import os
```

Qual o diretório que estamos trabalhando neste momento?

In [ ]:
```
os. getcwd()
```

Out[ ]:
```
'C:\\Users\\victo\\Formação em Dados'
```

In [ ]:
```
# Caso você estivesse trabalhando com o Jupyter, ou algum outro IDE na sua máquina
os.chdir('/tmp')
```

Como renomear e renomear colunas

In [ ]:
```
dados.rename(columns={'Date': 'Data', 'Adj Close': 'WEGE3'}, inplace=True)
```

In [ ]:
```
dados.head()
```

Out[ ]:

Date	Data	Open	High	Low	Close	WEGE3	Volume
2017-07-24	2017-07-24	7.392307	7.473076	7.261538	7.426923	7.051007	2638220.0
2017-07-25	2017-07-25	7.476923	7.476923	7.307692	7.369230	6.996235	2505880.0
2017-07-26	2017-07-26	7.323076	7.442307	7.276923	7.326923	6.956069	2112240.0
2017-07-27	2017-07-27	7.388461	7.419230	7.223076	7.269230	6.901296	1910480.0
2017-07-28	2017-07-28	7.261538	7.296153	7.165384	7.200000	6.835570	2866760.0

In [ ]:
```
# Removendo colunas

dados.drop(['Data', 'Volume'], axis = 1, inplace = True)
```

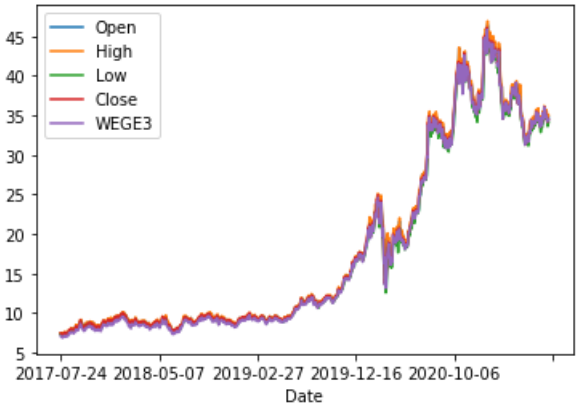
In [ ]:
```
dados.head()
```

Out[ ]:

Date	Open	High	Low	Close	WEGE3
2017-07-24	7.392307	7.473076	7.261538	7.426923	7.051007
2017-07-25	7.476923	7.476923	7.307692	7.369230	6.996235
2017-07-26	7.323076	7.442307	7.276923	7.326923	6.956069
2017-07-27	7.388461	7.419230	7.223076	7.269230	6.901296
2017-07-28	7.261538	7.296153	7.165384	7.200000	6.835570

In [ ]:
```
# Plotando todas as colunas

dados.plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x24cc8664ac0>
```



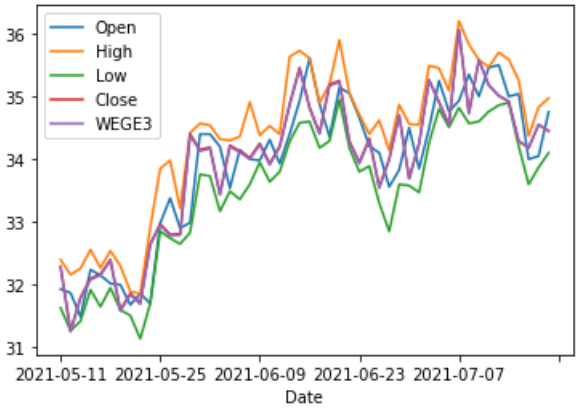
Quantas linhas e colunas tem o dataframe?

In [ ]:
```
dados.shape
```

Out[ ]:
```
(993, 5)
```

Filtrando apenas algumas linhas específicas

In [ ]:
```
dados.iloc[943:994].plot();
```



Aumentando o tamanho do gráfico

In [ ]:
```
dados.iloc[943:994].plot(figsize = (10,10));
```