MC102

Uma introdução ao Python

Bruno Antonio Tretto

1º Semestre-2025

Python

- Dicas:

- Para adicionar comentários no seu código usar #, o python não irá reconhecê-lo;
- Não usar barra de espaço nos nomes de variáveis, para isso utilizar _;
- Separador de casa decimal é o “ponto”, não é vírgula;
- Número inteiro=int;
- Número decimal=float;
- String é como fala de um texto;
- Booleana é verdadeiro ou falso;
- Símbolo diferente !=;
- Para fazer uma soma com o mesmo valor, como produtos = produtos + 1, podemos escrever produtos +=1, essa operação vale para qualquer outra, como *, /, -, %...

- Print:

-Vai mostrar a informação no terminal, escreve-se:

-Print(‘info’) → info

-Posso colocar também uma informação secundária, como:

print(‘Faturamento: ”, 1000); → Faturamento: 1000

-Por padrão o print separa por espaço e termina a linha, se quiser mudar usar sep=”:

Print (x, y, sep=’bruno’) → xbrunoy

-Para printar elementos separados por um espaço, ou vírgula:

print("bruno", *end* = "asd") → brunoasd

Uma forma mais fácil para colocar informações no print são as F strings, usa-se:

print(f’texto {x}”), onde x é uma variável do problema

-Posso usar mais de uma linha e quebrá-las, basta usar aspas triplas;

-Para usar uma quantidade definida de casas decimais:

suponha que quero a variável s com 2 casas decimais

print(f’ Salário = U$ {s:.2f}’);

- Operações:

Podemos fazer operações matemáticas básicas, como: Somar(+), subtrair(-), multiplicar(*), calcular mod(%), exponenciação(**), divisão inteira(//);

-Se eu quero só a parte inteira de uma divisão posso colocar int(148/12) ou 148//12;

-Para arredondar o número: round(número);

-Números não inteiros são caracterizados no python por “float”;

-Expressão do tipo x/y sempre resulta em float, mesma que seja um inteiro;

-Expressões x+y, x-y, x*y, x**y, x//y e x%y resultam em:

Int se x e y são ambos int

Float se x ou y são float;

- Variáveis:

Variáveis são nomes para regiões de memória do computador, elas servem para armazenar dados, como um nome próprio, bons nomes de variável deixam o código mais legível.

O nome variável vem do fato que ela pode mudar, se inicialmente tenho uma variável x com valor 10, posso em algum momento posterior do meu código redefinir x como 11.

Variáveis precisam começar com uma letra ou _, os outros caracteres podem ser letras ou números, elas são sensíveis a letras maiúsculas ou minúsculas.

Boas práticas de programação indicam que nomes significativos são bons nomes de variáveis(nome é melhor que n), mas nomes muito grandes devem ser evitados.

- Estruturas de decisão:

Basicamente as estruturas de decisão são: if, else e elif;

```
1 n = int(input("Entre com n: "))
2
3 if n % 2 == 0:
4         print(n, "é par")
5 else:
6         print(n, "é impar")
```

O python utiliza a indentação para criar blocos de código, o : ao final da linha indica que a condição do if/else acabou e que um bloco de código ira começar.
O if/else verificam o valor da variável booleana(verdade ou mentira) se for True executa o código do if e se for False executa a parte do else.

Os operadores lógicos do python são:

a	b	a and b	a or b	Not a
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

"a" e "b" podem ser qualquer expressão booleana, também pode-se escrever expressões do tipo a and not b, quando isso acontecer fica mais fácil usar parênteses: (a and (not b)).

- Útil lembrar das regras de Morgan:

not (a or b) é equivalente a (not a) and (not b);

not (a and b) é equivalente a (not a) or (not b);

- Comandos de repetição (While):

Ele repete o mesmo código até que algo aconteça, sua sintaxe é parecida com a do if, o ":" indica o começo do bloco e tem que ser indentado. Ex:

```
1   x = 1
2   while x != 0:
3       x = int(input("Digite um valor: "))
4   print("FIM")
5
```

Ao digitar qualquer valor o programa irá solicitar outro, mas ao digitar zero, o programa para;

- Listas:

São usadas para armazenar uma grande quantidade de dados, tantos quanto você queira. Para criar uma, usa-se: lista = [] ou lista = list()

Ex:

Lista = [1, 2, 3, 4, 7]

Lista = [x, y, z]

Para adicionarmos um valor "x" no final da lista, podemos usar o comando lista.append(x), sendo ele uma variável, uma constante, expressão ou outra lista.

Para acessar ou buscar dados da lista com um índice usaremos o comando: lista[0], o índice das listas sempre começa em 0, se o intuito for acessar uma lista de trás para a frente pode-se usar o comando lista[-1].

Aqui está um exemplo para estudo de um código que pede o número de elementos da lista, a seguir pede os elementos e os adiciona na lista e depois solicita que você digite um número e se ele está na lista ele printa True, caso contrário False:

```
n = int(input("Numero de elementos: "))
l = []
i = 0

while i < n:
    l.append(int(input("Entre com o número: ")))
    i += 1

k = int(input("k: "))

i = 0
Encontrou = False
while i < n:
    if l[i] == k:
        Encontrou = True
    i += 1

print (Encontrou)
```

Esse comando de procurar elementos em uma lista é tão comum que recebe uma sintaxe própria, ele é chamado de “in”, assim o comando anterior poderia ser substituído por:

```
n = int(input("Numero de elementos: "))
l = []
i = 0

while i < n:
    l.append(int(input("Entre com o número: ")))
    i += 1

k = int(input("k: "))

print(k in l)
```

-Para printar os elementos da lista l sem os colchetes, utilizar print(*l);

-Uma boa estratégia quando se tem um código para rodar x vezes e quando algo acontecer ele parar, pode-se usar a seguinte estrutura:
Começar com while True:
Escrever o código e colocar um if condição: break

- Range:

Ao invés de usar:

```
while i < n:
l.append(int(input("Entre com o número: ")))
i += 1
```

Podemos escrever:

```
For i in range(n):
    l.append(int(input("Entre com o número: ")))
```

O range(n) é como se fosse a lista [0, 1, 2, 3, ..., n-1] mas não é uma lista e pode ser convertido em uma usando list(range(n));

- Há três versões de range:

1. Range(fim):

Intervalo do tipo: [0,fim[;
Range(10) é 0, 1, 2, ..., 9;

2. Range(início, fim):

Intervalo do tipo: [início, fim[;
Range(2,10) é 2, 3, ..., 9;

3. Range(inicio, fim, passo):

Intervalo do tipo: [início, fim[pulando de passo em passo;
Range(10, 3, -2) é 10, 8, 6, 4

- Funções:

Todas as funcionalidades estudadas até agora são funções pré-definidas pelo python, podemos criar funções de acordo com nossas necessidades, otimizando códigos;
Para isso usa-se o comando def, da seguinte forma:

```
def lin():
    print('-------------')

lin()
```

Assim, toda vez que o comando lin() for mencionado lê retornará à definição e executará sem comando;

Também posso deixar uma variável na função, usando o exemplo superior:

```
def lin(msg):
    print('-------------')
    print(msg)
    print('-------------')

lin('Essa é a mensagem')
```

terminal →

```
-------------
Essa é a mensagem
-------------
```

As funções podem ser usadas com listas:

```
def dobra(lst):
    i = 0
    while i < len(lst):
        lst[i] *= 2
        i += 1

lista = [1, 2, 3, 4]
dobra(lista)

print(lista)
```

terminal → [2, 4, 6, 8]

Podemos deixar valores pré-estabelecidos para as funções, como no seguinte exemplo:

```
def somar(a = 0, b = 0, c = 0):
    s = a + b + c
    print(f"A soma vale {s}")


somar (1, 4, 5)
somar(2)
```

Assim caso só adicione uma variável na função, ela já tem um valor pré-estabelecido que é 0;

Atentar as variáveis definidas dentro das funções, caso isso ocorra dizemos que ela tem um escopo local e só funcionam dentro dela:

```
def teste(b):
    b += 3
    print(b)

a = 4
teste(a)
```

Mesmo com outro nome de variável, o Python executa o código com a variável global;

```
def teste(b):
    a = 5
    b += 3
    print(b)
    print(a)

a = 4
teste(a)
```

Se eu definir uma variável dentro da função o Python irá criar uma variável local e passará a usá-la, mas somente dentro da função, nesse exemplo o terminal:
7
5

```
def teste(b):
    global a
    a = 5
    b += 3
    print(b)
    print(a)

a = 4
teste(a)
```

Se usarmos o comando “global” o Python não vai criar nenhuma variável no escopo local e fará todos seus comandos com base na global, que agora para o resto do código passará a ser 5;

```
def soma(a, b):
    s = a + b
    return s

r1 = soma (2, 3)
print(r1)
```

As funções podem retornar ou não valores,
Assim, essa variável torna-se útil e posso realizar todas outras funções já conhecidas com ela;

Podemos criar funções sem definir em Python, são as famosas funções anônimas (funções Lambda), os dois códigos abaixo são idênticos, um usando a versão explicita da função e o outro usando a função lambda.

```
def calcular_imposto(a):
    return a * 0.3

a = 1000
print(calcular_imposto(a))
```

```
a = 1000
calcular_imposto = lambda a: a * 0.3
y = calcular_imposto(a)

print(y)
```

Um bom hábito é logo após a função descrever o que ela faz com ‘’’segue as instruções para uso da função’’’, assim quando outro programador for ler o código e precisar estudar sua função ele usará help(função) e aparecerá para ele o que você escreveu dentro dessas aspas como se fosse um manual;

Uma função muito útil em Python é Map, ele é usado como map(função, iterador) e aplica tal função a cada elemento desse iterador:

- Tabela ascii:

Binário	Decimal	Hexa	Glifo	Binário	Decimal	Hexa	Glifo	Binário	Decimal	Hexa	Glifo
0010 0000	32	20		0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p
0011 0001	49	31	1	0101 0001	81	51	Q	0111 0001	113	71	q
0011 0010	50	32	2	0101 0010	82	52	R	0111 0010	114	72	r

A tabela ascii é usada para facilitar python;
Trabalha-se com ela da seguinte forma:
-ord(a) = 61
-chr(61) = a

- Módulos e pacotes:

Dentro da linguagem python, posso usar o comando import para importar algo de uma "biblioteca" (conjunto de funções) para o meu código. Se quiser algo específico da biblioteca uso o comando from x import y.

```python
import math
num = int(input("digite um numero"))

raiz = math.sqrt(num)

print(raiz)
```

Uma biblioteca conhecida no Python é a math, segue um exemplo do uso de uma função presente nela, a sqrt.

No site python.org temos todas as bibliotecas e suas funcionalidades disponíveis, na pasta pypi temos as bibliotecas criadas por outros usuários.

Em um código com várias funções posso criar um outro arquivo na mesma pasta, como por exemplo a pasta uteis.py e colar todas as funções lá, se quiser usá-las posso escrever no meu código:

1. from uteis import x, y, z (importa função específica)

Nesse caso, posso escrever somente
Y = uteis(num)
E as determinadas funções do outro arquivo irão ser executadas.

2. import uteis (importa todas funções)

Nesse caso, para usar alguma função do arquivo uteis devo usar a seguinte sintaxe:

y = uteis.x(num)

Quando temos vários módulos em um código recorremos aos pacotes, eles são uma pasta que contém vários módulos separados por assuntos.

from uteis import datas

Para criar um pacote devo criar dentro do projeto uma pasta, dentro dessa pode haver várias outras pastas, uma para cada pacote. Existe uma sintaxe para nome de arquivos dentro de pacotes: __init__.py(Dentro de cada uma dessas pastas deve ter um arquivo com esse nome).

Um cuidado, quando o arquivo é importado o Python também o executa, por isso, se tiver comandos nele, esses serão executados também. Logo, é interessante guardar arquivos para usar como módulos somente com funções.

- Tratamento de erros e exceções:

Há diversos erros e exceções em Pyhton (NameError, ValueError, ZeroDivisonError, TypeError, IndexError, OSError, ...).

Um exemplo de código trabalhando com ele estará exemplificado abaixo:

```python
try:
    a = int(input('Numerador: '))
    b = int(input('Denominador: '))
    r = a / b

except (ValueError, TypeError):
    print('Tivemos um problema com os tipos de dados que você digitou.')
except ZeroDivisionError:
    print('Não é possível dividir um número por zero!')
except KeyboardInterrupt:
    print('O usuário preferiu não informar os dados!')
except Exception as erro:
    print(f'O erro encontrado foi {erro.__cause__}')

else:
    print(f'O resultado é {r:.1f}')

finally:
    print('Volte sempre! Muito obrigado!')
```

A sintaxe usada é try, o programa tentará rodar o que estará em try, se der erro ele irá para os except, onde está representado o que fazer para cada tipo, se não ocorrer nenhum deles o programa irá para o else e executará isso. A parte representada pelo finally ocorrerá sempre, dando erro ou não no programa. As cláusulas else e finally são opcionais.

```python
while True:
    n = int(input("Digite um número: "))

    if n < 0:
        raise ValueError("Número negativo")
```

Também podemos criar erros em python, para isso utiliza-se “raise” exemplificado no código abaixo, ele lê um número inteiro e cria um ValueError se esse for negativo.

- Listas (aprofundamento):

Nesse tópico aprofundaremos algumas coisas a respeito de listas, como:

- lista.clear(): Apaga todos os elementos da lista;

-len(lista): Devolve o número de elementos da lista;

-lista.insert(posição,variável): Adiciona na posição a variável e desloca todas outras uma posição;

-del lista[x]: Apaga o elemento da posição “x” na lista;

-lista.remove(x): Apaga o elemento “x” da lista, se esse elemento for repetido ele apagará somente sua primeira aparição;

-lanche.pop(x): Remove o elemento da posição “x” da lista contada de trás para a frente;

-lista.sort(): Ordena os elementos da lista em ordem crescente;

-lista.index(elemento): Devolve o primeiro índice i da lista tal que lista[i] == elemento e lança ValueError se esse elemento não existir;

-lista.reverse(): Inverte os valores da lista;

-lista.sort(reverse=True): Ordena os elementos da lista em ordem decrescente;

-lista.copy(): Devolve uma cópia da lista;

-lista.extend(lista2): Adiciona os elementos da lista2 na lista;

-Podemos fatiar(slice) uma lista:

l[i:f] nos dá os elementos l[i], l[i + 1], …, l[f - 1]
- l[:f] - até o elemento f - 1 (omitindo i)
- l[i:] - todos os elementos a partir do i (omitindo f)
- l[:] - toda a lista (omitindo i e f)
-l[i:f:p] nos dá os elementos l[r], onde r = i + k * p com k inteiro não negativo e r < f (como no range)

-Se eu tenho uma lista [5, 9, 4] e quero printar seus elementos separados de três pontos:

```
lista =[4, 5, 9]

for v in lista:
    print(f"{v}...", end = '')
```

- Se eu tenho uma lista a e eu a igualo a uma lista b o Python entenderá as duas como a mesma coisa em todo o seu código, se eu quero copiar a lista a devo "fatiar" ela escrevendo a = b[:];

-Se eu quiser adicionar uma lista em outra devo usar um append com o fatiamento também:

Lista2.append(lista1[:]), ele adiciona a lista 1 na posição final da lista 2

Se eu quero printar o primeiro elemento da lista 1, que agora está na posição x da lista 2 eu faço:

Print(lista2[x][1])

- Strings aprofundamento:

A classe string é muito parecida com a classe list.
-Você pode acessar os caracteres de uma string usando índice:

```
y = "palavra"
print(y[0])
```

→ Terminal: p

-Pode-se escrever "for letra in string" para percorrer todas as letras da string;
-Pode usar slices de string;
-NÃO dá pra alterar caracteres de string;
-Pode remover caracteres com del;
- Se eu quero escrever uma string que tem ‘ ou " devo usar uma barra antes, assim o Python vai interpretar ‘como um carácter e não como final de String;
-A barra invertida "\" transforma caracteres especiais em normais:
Se eu quero representar um espaço eu uso /n;

Se eu quero representar um tab eu uso /t;
-Podemos comparar strings podendo usar <=, =, >, == e !=:
"ana" < "beto"
"ana" < "anamaria"
[1, 2, 3] < [1, 2, 4]
[1, 2, 3, 5] < [1, 2, 4]
[1, 2] < [1, 2, 4]

-sep.join(lista): Concatena uma lista usando sep como separador

```
y = ':'.join(["q", "n", "g"])

print(y)
```

→ Terminal: q:n:g

-s.split(sep): Quebra string s em cada ocorrência de sep em uma lista:

```
y = "palavra".split("a")

print(y)
```

→ Terminal: ['p', 'l', 'vr', '']

```
nomes = input("Entre com o seu nome completo: ").split(" ")

print(nomes)
```

→ Terminal: ['Bruno', 'Antonio', 'Tretto']
Cria uma lista com o separador que eu escolhi, nesse caso o "espaço";

- Dicionários:

Estruturas de dados semelhantes as tuplas e listas, com índices literais/personalizáveis.
- São identificados por {}: dados = {} ou dados = dict();
Exemplo:

```
dados = dict()

dados = { 'nome':'Pedro', "idade":25 }

print(dados["nome"])
print(dados["idade"])
```

→

```
Pedro
25
```

Para adicionar mais um elemento, como no exemplo acima adicionar o sexo, basta escrever:
dados['sexo']='M'
Não precisamos dar .append em dicionários
- Para remover elementos em um dicionário uso o comando del:
Del dados['idade']
Ele perde o elemento e seu valor;

- Para começar uma lista basta abrir e fechar chaves, não precisa ser na mesma linha:

```
filme = {'titulo':'Star Wars',
         'ano':1977,
         'diretor':'George Lucas'
}
```

No exemplo acima “título”, “ano” e “diretor” são chamados de Keys;

```
print(filme.values())
print(filme.keys())
print(filme.items())
```

→

```
dict_values(['Star Wars', 1977, 'George Lucas'])
dict_keys(['titulo', 'ano', 'diretor'])
dict_items([('titulo', 'Star Wars'), ('ano', 1977), ('diretor', 'George Lucas')])
```

```
for k,v in filme.items():
    print(f"o {k} é {v}")
```

→

```
o titulo é Star Wars
o ano é 1977
o diretor é George Lucas
```

-Posso misturar listas e dicionários:

```
locadora = [{'titulo':'Star Wars',
         'ano':1977,
         'diretor':'George Lucas'
},
{'titulo':'Avengers',
         'ano':2012,
         'diretor':'Joss Whedon'
} ]

print(locadora[0]['ano'])
print(locadora[1]['ano'])
```

→

```
1977
2012
```

- Conjuntos:

  Conjuntos são coleções não ordenadas de valores únicos, usadas para armazenar múltiplos itens, similares a duplas e listas;

-São criados com chaves, ordem não importa, são mutáveis;
-Não há sintaxe de acesso a elementos (conjunto não ordenado);
Alguns comandos:

```python
a = {1, 2, 3}
b = {3, 4, 5}

# Métodos de modificação
a.add(4)                # Adiciona o elemento 4 ao conjunto 'a'

a.remove(2)             # Remove o elemento 2 (erro se não existir)

a.discard(10)           # Tenta remover o elemento 10 (sem erro se não existir)

elemento_removido = a.pop()  # Remove e retorna um elemento arbitrário

a.clear()               # Remove todos os elementos de 'a'


print("union:", a.union(b))                  # União: {1, 2, 3, 4, 5}
print("intersection:", a.intersection(b))    # Interseção: {3}
print("difference:", a.difference(b))        # Diferença: {1, 2}

# Métodos de comparação
print("issubset:", {1, 2}.issubset(a))       # True, pois {1, 2} está contido em 'a'
print("issuperset:", a.issuperset({1, 2}))   # True, 'a' contém {1, 2}
print("isdisjoint:", a.isdisjoint({4, 5}))   # True, não há interseção entre a e {4, 5}
print("symmetric_difference:", a.symmetric_difference(b))  # {1, 2, 4, 5}
```

-sorted(conjunto) coloca ele em ordem alfabética ou crescente;

- Matrizes:

Em python, matrizes nada mais são do que listas de listas, isto é podemos representar a matriz abaixo:

$$\begin{bmatrix} 7 & 0 & 2 & 3 \\ 3 & 1 & 4 & 2 \\ 0 & 3 & 2 & 7 \end{bmatrix}$$

Simplesmente por m = [[7, 0, 2, 3], [3, 1, 4, 2], [0, 3, 2, 7]], isto é uma lista de linhas da matriz, onde cada linha é, também, uma lista.

-A célula da linha i coluna j é acessada escrevendo m[i][j], ex m[0][0] é 7;

-Matrizes podem ter mais dimensões, imagens são compostas por matrizes de duas dimensões, mas vídeos são sequências de imagens e, portanto, tem mais de 2D. Quando tinha 1 dimensão usava-se listas, quanto tinha 2 dimensões usava-se listas de listas, com 3 usa-se listas de listas de listas e assim por diante;
-Às vezes é necessário linearizar índices, na matriz apresentada acima a sua versão linearizada seria: l = [7 0 2 3 1 4 0 3 2];
(0, 0) deve ir para o índice 0
(0, 1) deve ir para o índice 1
(0, m − 1) deve ir para o índice m – 1
(1, 0) deve ir para o índice m
(1, 1) deve ir para o índice m + 1
(1, m − 1) deve ir para o índice 2m – 1
(i, j) deve ir para o índice i · m + j

- Compreensão de listas:

É basicamente uma sintaxe do Python para criar listas rapidamente
Ex:
[i for i in range(10)] → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[i ** 2 for i in range(10)] → [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Sintaxe: [exp for var in iter]
• exp é alguma expressão que pode usar o valor de var ;
• iter é algum objeto que pode ser iterado(listas, dicionários, conjuntos, ranges, entre outros);

- Podemos também filtrar os valores antes de mapear com a sintaxe
Sintaxe: [exp for var in iter if cond]
Ex:
[x for x in lista if x % 2 == 0] – Lista dos números pares de lista
[i ** 2 for i in range(10) if i % 2 == 0] → [0, 4, 16, 36, 64]

-Você também pode criar conjuntos e dicionários:
• {math.fabs(x) for x in lista} → {0, 1, 2, 3}
• {i: math.factorial(i) for i in range(5)} → {0: 1, 1: 1, 2: 2, 3: 6, 4: 24}

- Classes:

Classes são objetos em Python, define-se que nomes de Classes tenham iniciais maiúsculas nessa linguagem e que não se dê espaços por _ somente concatena as palavras;

- Ao criar uma classe em python inicia-se criando uma função "__init__()" que vai iniciar a classe, nela deve conter todas as características da minha classe.
Ao criar a função __init__ deve-se colocar o "self" como primeiro parâmetro, seguidos dos demais:

Class Exemplo:
```
def __init__(self, p1, p2, p3):
    self.p1 = "x"
    self.p2 = "y"
    self.p3 = "z"
```

Assim, no meu Código ao criar o objeto eu o referencio como:
objeto1 = Objeto(valor1, valor2, valor3)

Exemplo dado em aula:

```
class Estudante:
    def __init__(self, nome, RA, curso, nota):
        self.nome = nome
        self.RA = RA
        self.curso = curso
        self.nota = nota



ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
print(ana.nome, ana.RA, ana.curso, ana.nota)
```

→ Ana 123456 42 10.0
Nesse exemplo dizemos que Ana é um objeto da classe estudante e é uma instância de estudante;

-Podemos adicionar mais funções nas classes criadas, exemplo:

```
class Estudante:
    def __init__(self, nome, RA, curso, nota):
        self.nome = nome
        self.RA = RA
        self.curso = curso
        self.nota = nota

    def aprovado(self):
        return self.nota >= 5.0



ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)

if ana.aprovado():
    print(ana.nome, "esta aprovado")
else:
    print(ana.nome, "está reprovado")
```

→ Ana está aprovado

Importante notar que não foi passado parâmetro para ana.aprovado(), o Python já reconhece que self é ana.

Ao invés de usar o seguinte código acrescentado na classe acima para imprimir o estudante:

```
def imprime(self):
    print("RA:", self.RA,
    "Nome:", self.nome,
    "Curso:", self.curso,
    "Nota:", self.nota)
```

O python nos permite usar o método mágico "__str__" da seguinte forma:

```
def __str__(self):
    return (f"RA: {self.RA} Nome: {self.nome}" +
    f"Curso: {self.curso} Nota: {self.nota}")
```

Para simplificar ainda mais a escrito o Python aderiu os Dataclasses, são usados da seguinte forma:

```
from dataclasses import dataclass

@dataclass # decorador para a classe estudante
class Estudante:
    nome: str
    RA: int
    curso: int
    nota: float

    def aprovado(self):

        return self.nota >= 5.0

ana = Estudante(nome="Ana", RA=123456, curso=42, nota=10.0)
print(ana)
print(ana.curso)
```

O terminal devolverá:

Estudante(nome='Ana', RA=123456, curso=42, nota=10.0)

42

Basicamente, o uso de dataclasses te dá já pronto algumas coisas, como:
Métodos __init__, __repr__, __eq__...

Podemos comentar classes, do mesmo modo que é feito com funções, iniciando com três aspas simples e fechando com as mesmas três aspas.
Outro exemplo:

```
class Retangulo:
    def __init__(self, altura, largura):
        self.largura = largura
        self. altura = altura

    def area(self):
        return self.largura * self.altura

    def perimetro(self):
        return 2 * (self.largura + self.altura)

    def __str__(self):
        return "Retângulo " + str(self.largura) + "x" + str(self.altura)

r1 = Retangulo(10,3)
print(r1.area(), r1.perimetro())
```

```
from dataclasses import dataclass


@dataclass
class Retangulo:
    altura : float
    largura : float

    def area(self):
        return self.largura * self.altura

    def perimetro(self):
        return 2 * (self.largura + self.altura)

    def __str__(self):
        return "Retângulo " + str(self.largura) + "x" + str(self.altura)

r1 = Retangulo(10,3)
print(r1.area(), r1.perimetro())
```

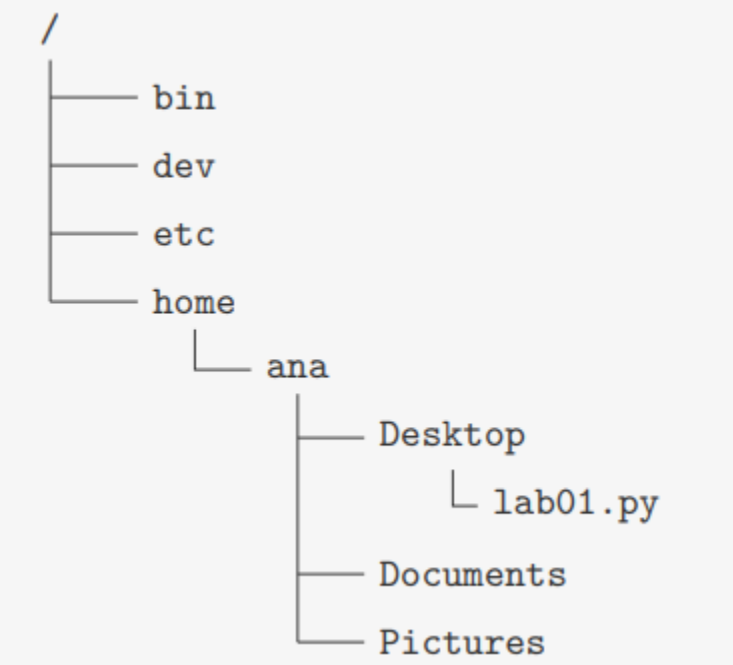
Os dois códigos irão lançar no terminal 30 26

Obs: Convenciona-se em Python que nomes de classes são dados por letra maiúscula;

- Arquivos:

Arquivos são formas de armazenar dados, blocos de armazenamento permanente, eles podem estar no HD, SD, pendrive... e são acessados logicamente por um sistema de arquivos. Esse sistema de arquivos é uma estrutura de acesso, arquivos não têm nome, mas sim endereço, arquivos são armazenados e diretórios, esses podem conter vários arquivos ou outros diretórios.

O sistema tem um ponto de origem, chamado raiz, no Windows cada dispositivo tem sua própria raiz.



Exemplo de um sistema Unix (Linux, macOS):
O caminho é absoluto, ele nos é dado desde a raiz" \".
No Windows seria algo do tipo:
C:\Usuarios\ana\Desktop\lab01.py

Caminho:
/home/ana/Desktop/lab01.py

Como há diferenças entre os módulos do Windows e do Linux, existe um padrão que funciona para ambos, o padrão os:

os.sep diz qual é o separador: '/' ou '\
os.path.join é útil para construir um caminho
– os.path.join('ana', 'Desktop', 'lab01.py')
– Linux/MacOS: ana/Desktop/lab01.py
– Windows: ana\Desktop\lab01.py

É uma boa prática de programação usar o os porque você programa independentemente do sistema operacional.

Alguns métodos úteis para os:

-os.chdir: mudar o diretório atual;
-os.mkdir: criar um diretório;
-os.remove: remover um arquivo (cuidado!);
-os.rename: renomear arquivo/diretório;
-os.scandir: pegar os arquivos e diretórios de um diretório;
-os.path.exists: verifica se um caminho existe;
-os.path.isdir: verifica se um caminho representa um diretório;

Para ler um arquivo em Python, precisamos abri-lo (solicitando permissão ao sistema operacional), ler seu conteúdo (letra a letra, linha a linha, ou todo de uma vez) e fechar ao arquivo (informando ao sistema operacional que não vamos mais utilizá-lo.

```
arquivo = open("arq.txt") # abre arq.txt do diretório atual
s = arquivo.read() # lê todo o conteúdo
print(s)
arquivo.close() # fecha o arquivo
```

Assim, o arquivo será lido do começo até o final (EOF — End of File).
-arquivo.read(): lê todo restante do arquivo;
-arquivo.read(k): lê os próximo k caracteres do arquivo (Se tiver menos do que k, lê menos);
- arquivo.readline(): lê até a próxima quebra de linha (\n);

Também é possível ler o arquivo linha a linha usando for:

```
arquivo = open("arq.txt")

for linha in arquivo:
    print(linha)

arquivo.close()
```

Para abrir um arquivo para escrita, temos alguns modos:
-'r' read leitura (padrão);
-'w' write escrita (apaga o conteúdo atual);
- 'a' append acréscimo;
- 'x' new file escrita apenas em arquivo novo;
Ex: arquivo = open('arq.txt', 'w')

Existem também formas de abrir um arquivo para leitura e escrita simultânea: r+, w+, a+ (Nesse caso, você precisará andar pelo arquivo usando o método seek);

Duas formas de escrever:
-arquivo.write(texto): recebe uma string texto e escreve no arquivo (não quebra linha automaticamente com o print);
-arquivo.writelines(lista): escreve as strings de lista no arquivo(não quebra linha automaticamente com o print);

```
arquivo = open("arquivo.txt", "w")
arquivo.write("Olá, Mundo!\n")
arquivo.close()
```

Dados escritos podem ficar na memória até fechar o arquivo, eles são salvos no arquivo quando esse é fechado ou se der o comando arquivo.flush();

Se o seu programa tiver uma exception, o arquivo pode não ser fechado e o conteúdo não pode ser salvo, você pode atingir o limite de arquivos abertos, seu programa pode ficar muito lento por consumo de código, para isso utiliza-se:

```python
with open("arquivo.txt", "w") as arquivo:
    arquivo.write("Olá, Mundo!\n")
```

Quando o bloco acaba, o arquivo é fechado (mesmo com exception).

- Algoritmos de ordenação:

1)Selection sort:

A intenção é olhar para uma lista de dados e ordenar qual o maior ou menor.

No algoritmo por seleção assume-se o primeiro elemento como mínimo e, posteriormente, compara-se todos os próximos elementos da lista com ele, se o próximo for menor/maior atualiza-se o valor de mínimo ou máximo.

Ex:

```python
lista = [1, 2, 5,-1,  7, 9, 3]

menor = lista[0]

for i in range(1, len(lista)):
    if lista[i] < menor:
        menor = lista[i]

print(menor)
```

Para somente saber qual o menor elemento.

Para ordenar os elementos:

```python
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        menor_indice = i
        for j in range(i+1, n):
            if lista[j] < lista[menor_indice]:
                menor_indice = j
        lista[i], lista[menor_indice] = lista[menor_indice], lista[i]
```

2)Bubble sort:

A intenção é olhar para uma lista de dados comparando elementos de posições consecutivas, primeiramente compara-se o elemento da posição 0 com o da posição 1, se o da 1 for menor troca eles, se não nada é feito.

```python
def bubble_sort(lista):
    n = len(lista)
    for i in range(n-1):
        for j in range(n-1 - i):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

print(bubble_sort([1, 3, 6, -1, 0, 29, 4]))
```

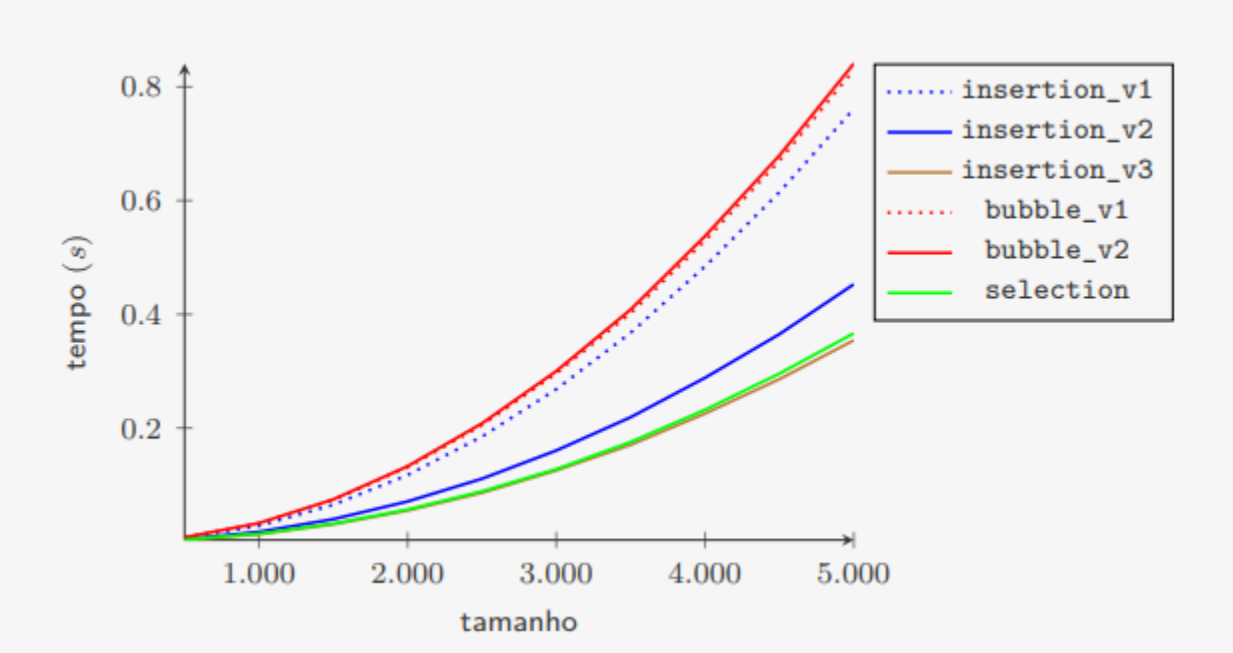
3)Insertion sort:

Ele também compara em duplas, começando com o elemento 0 e a partir dele comparando todos os próximos com os que estão imediatamente a sua esquerda, se ele for maior, troca-se ambos de posição.

```python
def insertion_sort(lista):
    n = len(lista)
    for i in range(1, n): #Podemos "ignorar" o primeiro já que ele já está ordenado
        chave = lista[i]
        j = i - 1

        while j >=0 and lista[j] > chave:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j+1] = chave
    return lista
```

Comparação entre os tempos dos 3 algoritmos:



- Algoritmos de busca:

Dada uma lista l e um número x, o algoritmo quer encontrar, se existir, um índice i tal que l[i] = x. A ideia do algoritmo é percorrer a lista do início para o fim, procurando x, se encontrar devolve-se x ande está e se não x não está na lista:

Pseudocódigo:

```
1 BuscaSequencial(l, x)
2         Seja n o tamanho de l
3          Para i = 0 até n - 1
4                 Se l[i] = x
5                         Devolva i
6         Devolva -1
```

1)Busca sequencial:

Seus passos são simples e ele claramente sempre termina. Se a primeira ocorrência de x em l é a posição k, ele devolve k, se x não está em l a linha 4 sempre falha e devolve -1

```python
def busca_sequencial(lista, alvo):
    for i in range(len(lista)):
        if lista[i] == alvo:
            return i  # Retorna o índice onde o alvo foi encontrado
    return -1  # Retorna -1 se o alvo não estiver na lista
```

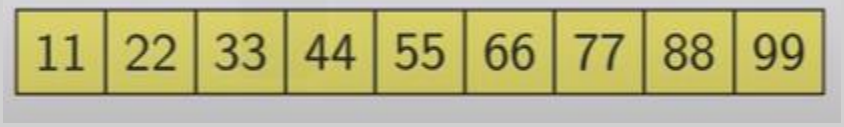
2)Busca ordenada(binária):

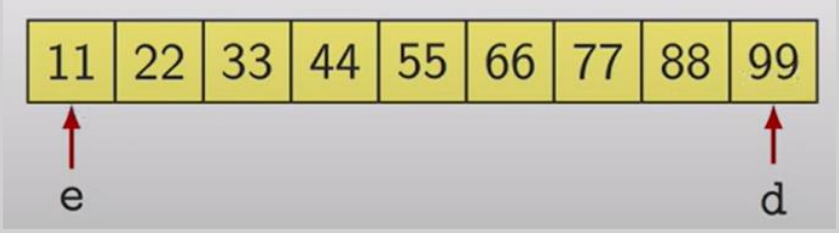
Ele já recebe uma lista ordenada, ao invés de olhar a lista toda podemos pensar em otimizar o código olhando para parte dela:

Pseudocódigo:

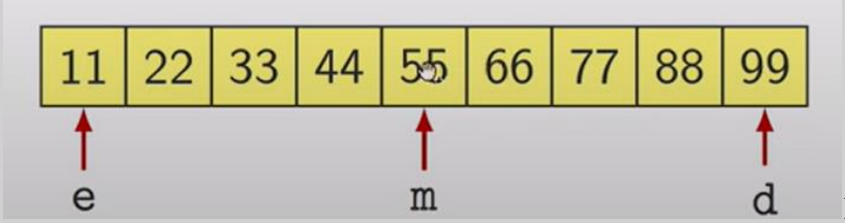
```
1 BuscaBinaria(l, x)
2       Seja n o tamanho de l
3       e = 0                                   #e de esquerda
4       d = n – 1                               #d d direita
5       Enquanto e <= d
6               m = (e + d) / 2 # divisão inteira
7               Se l[m] == x
8                       Devolva m
9               Se l[m] < x # está para a direita
10                      e = m + 1
11              Se l[m] > x # está para a esquerda
12                      d = m - 1
13       Devolva -1
```

Buscando por 33:

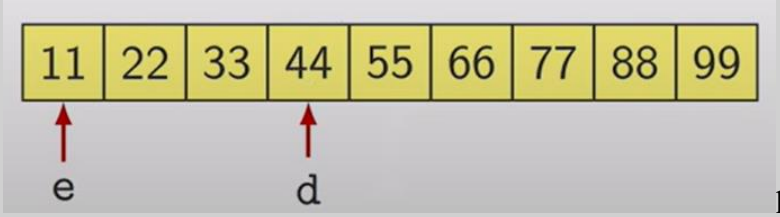




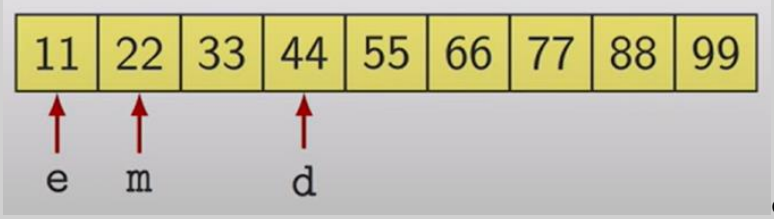
linhas 3 e 4



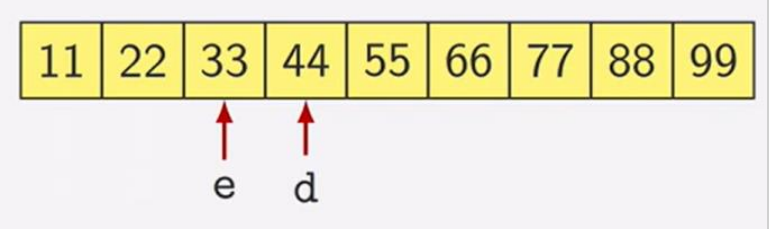
linha 6

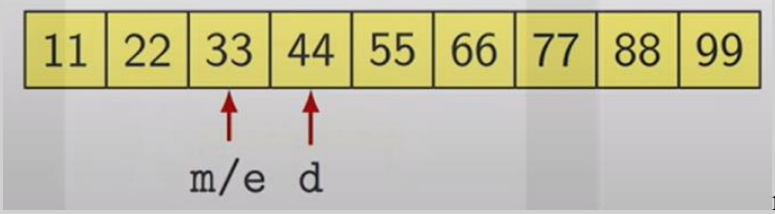


linha 9

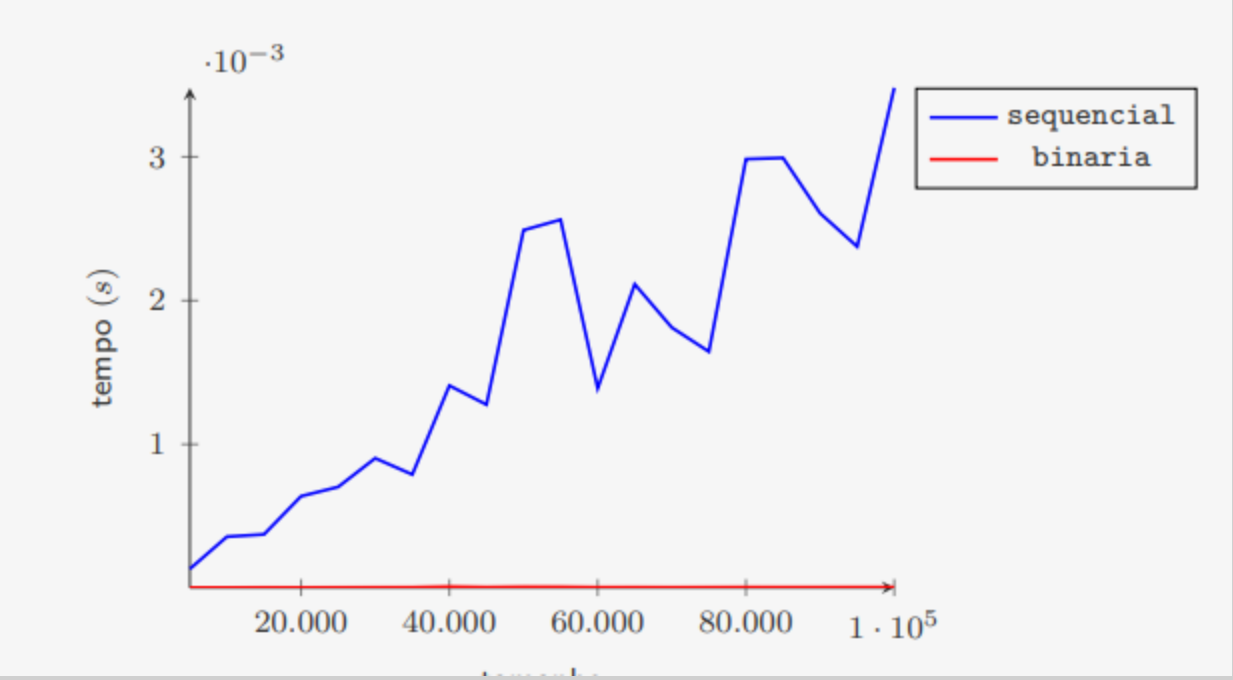


calcula-se o m novamente





m = e



Tempo busca binária x sequencial

- Recursão:

Recursão é o conceito de uma função chamar a si mesma para resolver algum problema computacional a partir de instâncias pequenas (casos base) e maiores (casos gerais). Como exemplo temos a função fatorial:

```
1 def fatorial(n):
2        if n == 0:
3                return 1
4        else:
5                return n * fatorial(n - 1)
```

O Python sabe a linha de código que fez a chamada de função, isso gera uma pilha de chamadas, para calcular o fatorial de 4 ele vai para o de 3, a seguir o de dois que cai no caso base 1 e volta até o 4, retornando o resultado.

Podemos usar a recursão para ordenar elementos:

1)MergeSort:

Uma lista maior de n elementos é dividida em duas menores, essas por suas vezes são divididas pela metade de novo, até que existam n listas com um só elemento, essas por sua vez já estão ordenadas. A seguir, começa o processo contrário, onde junta-se elementos dois a dois e ordena-os, posteriormente junta 4 em 4, ordenando-os até chegar na lista principal, todos esses processos sendo realizados recursivamente.

```
1 def merge(lista, esquerda , meio, direita):
2       aux = []
3       i, j = esquerda , meio + 1
4       while i <= meio and j <= direita:
5               if lista[i] <= lista[j]:
6                       aux.append(lista[i])
7                        i += 1
8               else:
9                       aux.append(lista[j])
10                       j += 1
11      while i <= meio: # Copia o restante da primeira metade
12              aux.append(lista[i])
13               i += 1
14      while j <= direita: # Copia o restante da segunda metade
15              aux.append(lista[j])
16               j += 1
17       for i in range(esquerda , direita + 1): # Copia de volta
18              lista[i] = aux[i - esquerda]
```

Ordenação: receberemos uma faixa da lista, começando na posição esquerda e terminando na posição direita, dividimos a faixa em duas, o caso base é uma faixa de tamanho 0 ou 1, já ordenada!!

```
1 def mergesort(lista, esquerda , direita):
2       if esquerda < direita:
3               meio = (esquerda + direita) // 2
4               mergesort(lista, esquerda , meio)
5               mergesort(lista, meio + 1, direita)
6               merge(lista, esquerda , meio, direita)
```

1)QuickSort:

A ideia base é escolher um elemento, nesse caso chamado de pivô, e deixar a direita dele somente algarismos que são maiores que ele e a esquerda somente algarismos que são menores que ele, isso é feito para uma quantidade suficiente de elementos da lista para garantir que todos estão ordenados.

```
1 def quicksort(lista, esquerda , direita):
2       if esquerda < direita:
3               k = partition(lista, esquerda , direita)
4               quicksort(lista, esquerda , k - 1)
5               quicksort(lista, k + 1, direita)
```

Agora, basta particionar a lista em duas e ordenar os lados esquerdo e direito, para particionar andamos da direita para a esquerda com um índice i. De i até pos – 1 ficam os menores que o pivô e de pos até direita ficam os maiores ou iguais ao pivô.

```
1 def partition(lista, esquerda , direita):
2         pivo = lista[esquerda]
3         pos = direita + 1
4         for i in range(direita , esquerda - 1, -1):
5                 if lista[i] >= pivo:
6                         pos -= 1
7                          lista[i], lista[pos] = lista[pos], lista[i]
8         return pos
```

Comparação entre os tempos dos algoritmos: